Introduction to Programming Concepts with MATLAB

2022-10-02

Contents

Front Cover **22**

Information **25**

License **27**

Dedication **29**

Thoughts **31**

Author Bios **33**

Preface **37**

Contact Information **39**

Acknowledgements **41**

What is New With This Edition **43**

A Note to Students **45**

A Note to Instructors **47**

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.1 – MATLAB Introduction . . . .

Learning Objectives . . . .

What is MATLAB? . . . .

What is MATLAB used for in engineering and science? . . . .
How can I get MATLAB onto my computer? . . . .
Are there any free alternatives to MATLAB? . . . .
Where can I find more information and help with MATLAB online? .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.2 – Hello World . . . .
Learning Objectives . . . .
Where can I find and open the MATLAB program? . . . .
Step 1: Create a New m-file . . . .
Step 2: Write the 'Hello World' Code . . . .
Step 3: Run the Program . . . .
Step 4: Make Your Program a Little Fancier . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.3 – MATLAB Environment . . . .
Learning Objectives . . . .
MATLAB Environment Windows and Parts . . . .
Navigation Ribbon . . . .
Working Folder Location . . . .
Current Folder . . . .
Command Window . . . .
Editor Window . . . .
Workspace . . . .
Status Bar . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.4 – Changing MATLAB Preferences . . . .

Learning Objectives . . . .

How can I change the window layout in MATLAB? . . . .

Changing Basic User Preferences . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.5 – The m-file . . . .

Learning Objectives . . . .

What is an m-file? . . . .

How do I save my m-file? . . . .

How do I input variables and expressions into the m-file? . . . .

How do I run the m-file? . . . .

What are the clc and clear commands? . . . .

How can I place comments in my m-file? . . . .

Can I separate my code into parts within the m-file? . . . .

What does the color highlighting in the m-file mean? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.6 – The Command Window . . . .

Learning Objectives . . . .

What is the Command Window and how can I use it? . . . .

How can I suppress outputs in the Command Window? . . . .

Can I view help from the Command Window? . . . .

Can the Command Window do it all? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.7 – Publishing an m-file . . . .

Learning Objectives . . . .

What does publishing do? . . . .

How can I publish in MATLAB? . . . .

How can I get a PDF file of my published code? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.1 – Variables and Naming Rules . . . .

Learning Objectives . . . .

What is a mathematical variable? . . . .

What is a programming variable? . . . .

How can I give my results a variable name of their own? . . . .

What are some possible problems with naming an expression? . . . . .

Are there benefits to good practices for variable naming? . . . .

Are there some guidelines for variable naming that I can follow? . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.2 – Characters and Strings . . . .

Learning Objectives . . . .

What is a character? . . . .

What is a string? . . . .

What makes characters/strings special? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.3 – Working with Strings . . . .
Learning Objectives . . . .
How do I join two strings together? . . . .
How do I search and count strings? . . . .
How do I make a whole string lower or upper case? . . . .
Can I split a string into its component pieces? . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.4 – Inputs and Outputs . . . .
Learning Objectives . . . .
How can I get input from the user? . . . .
How do I display notes in the Command Window? . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.5 – Data Types . . . .
Learning Objectives . . . .
What is a data type? . . . .
What are the MATLAB data types? . . . .
Why are data types important? . . . .
How do I check the data type of a variable? . . . .
Can I convert between data types? . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.6 – Vectors and Matrices . . . .
Learning Objectives . . . .
Why is the program called MATLAB? . . . .
What is a matrix? . . . .
Do I need to know any special types of matrices? . . . .
What is a vector? . . . .
How do I define a vector or a matrix in MATLAB? . . . .
What are some basic functions and commands for matrix manipulation?
How can I reference strings as vectors? . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.7 – How to Debug Code . . . .
Learning Objectives . . . .
What is an error? . . . .
What is a warning? . . . .
How can I solve the problem with my code? . . . .
Can I pause my program part of the way through? . . . .
What if I cannot find the exact place of the error? . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 3: PLOTTING

Lesson 3.1 – Plots and Figures . . . .
Learning Objectives . . . .
How can I visualize (plot) data in MATLAB? . . . .
How do I plot data pairs (points) in MATLAB? . . . .
How do I plot a function in MATLAB? . . . .
What is the difference between a figure and a plot? . . . .

How can I enter nonlinear functions for plotting? . . . .
What are some possible errors with plotting? . . . .
How do I show multiple data sets on the same plot? . . . .
What are some other types of plots that MATLAB can generate? . . .
How do I plot on more than one figure in the same m-file? . . . .
What is the close command? . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 3: PLOTTING

Lesson 3.2 – Plot Formatting . . . .
Learning Objectives . . . .
How can my MATLAB graph look nicer? . . . .
What are some terms I should know for plots? . . . .
How can I change the color and style of lines and markers on a plot? .
How can I make the function and points on the graph look nicer? . . .
How can I put a title and axis labels on my plot? . . . .
How can I add a legend to my plot? . . . .
How can I add a grid to my graph? . . . .
How can I add special characters in my axis labels and title? . . . . .
How can I change axis limits and tick labels? . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 3: PLOTTING

Lesson 3.3 – Advanced Plotting . . . .
Learning Objectives . . . .
Does MATLAB have more plotting capabilities? . . . .
How can I create a bar graph? . . . .
How can I create a 3D line plot? . . . .

How can I create a 3D surface plot? . . . .

How can I create a polar plot? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.1: Basic Algebra, Logarithms, and Trigonometry . . . .

Learning Objectives . . . .

What kind of mathematical functions and operations are available in MATLAB? . . . .

How do I use logarithmic functions in MATLAB? . . . .

What about a logarithm that is not natural? . . . .

How can MATLAB evaluate trigonometric functions? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.2: Symbolic Variables . . . .

Learning Objectives . . . .

What is a symbolic variable? . . . .

What is a MATLAB toolbox? . . . .

How do I use symbolic variables? . . . .

How do I clear specific variables? . . . .

How can I convert from syms data type to other data types? . . . . .

Can I replace a symbolic variable with a value? . . . .

How can I change the output format of syms? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.3: Solution of Linear and Nonlinear Equations . . . .
Learning Objectives . . . .
How do I solve for roots of a linear equation? . . . .
What is a nonlinear equation? . . . .
How can I use MATLAB to solve nonlinear equations? . . . .
Is there a faster way to work with polynomial equations in MATLAB?
Can I plot with symbolic variables? . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.4: Differential Calculus . . . .
Learning Objectives . . . .
What is a derivative? . . . .
How do I take the derivative of a function in MATLAB? . . . .
Where are derivatives used in engineering? . . . .
How do I find the derivative of a discrete function in MATLAB? . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.5: Integral Calculus . . . .
Learning Objectives . . . .
What is integration? . . . .
How does MATLAB conduct symbolic integration? . . . .
Can MATLAB do numerical integration of discrete functions? . . . . .
Example 2 . . . .
Lesson Summary of New Syntax and Programming Tools . . . .
Multiple Choice Quiz . . . .
Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.6 – Linear Algebra . . . .

Learning Objectives . . . .

What is linear algebra? . . . .

How do I add and subtract matrices? . . . .

Can I use math functions like `sin()` on matrices? . . . .

How do I perform matrix multiplication? . . . .

What is the difference between matrix and array operations? . . . . .

How do I take the inverse of a matrix? . . . .

Can MATLAB do advanced matrix and vector operations? . . . . . .

How can I solve systems of equations with MATLAB? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.7 – Curve Fitting . . . .

Learning Objectives . . . .

What is curve fitting? . . . .

What is interpolation? . . . .

How can I interpolate data in MATLAB? . . . .

What is spline interpolation? . . . .

How do I conduct spline interpolation? . . . .

What is regression? . . . .

How do I do regression in MATLAB? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.8 – Curve Fitting: Plotting the Results . . . .

Learning Objectives . . . .

How can I plot the results of curve fitting? . . . .

What are some common mistakes when plotting curve fitting results?

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 4: MATH AND DATA ANALYSIS

Lesson 4.9 – Ordinary Differential Equations . . . .

Learning Objectives . . . .

What is a differential equation? . . . .

How do I set up and solve a differential equation? . . . .

How do I solve a higher order ODE? . . . .

What are the limitations of using the `dsolve()` function? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 5: CONDITIONAL STATEMENTS

Lesson 5.1 – Conditions and Boolean Logic . . . .

Learning Objectives . . . .

What are conditions? . . . .

What is Boolean logic? . . . .

Can different data types be identified in MATLAB? . . . .

How can I round numbers in MATLAB? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 5: CONDITIONAL STATEMENTS

Lesson 5.2 – Conditional Statements: if and if-else . . . .

Learning Objectives . . . .

What is a conditional statement? . . . .

What is the if statement? . . . .

What is the if-else statement? . . . .

Can I use multiple conditions in a single expression? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 5: CONDITIONAL STATEMENTS

Lesson 5.3 – Conditional Statements: if-elseif . . . .

Learning Objectives . . . .

What is the if-elseif statement? . . . .

Independent vs. Dependent Cases . . . .

What is the if-elseif-else statement? . . . .

What is the difference between the else and elseif conditional clauses?

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 6: PROGRAM DESIGN AND COMMUNICATION

Lesson 6.1 – Flowcharts . . . .

Learning Objectives . . . .

What is a flowchart? . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 6: PROGRAM DESIGN AND COMMUNICATION

Lesson 6.2 – Pseudocode . . . .

Learning Objectives . . . .

What is a pseudocode? . . . .

How are pseudocodes used? . . . .

How can I convert a pseudocode for a problem into a program? . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 6: PROGRAM DESIGN AND COMMUNICATION

Lesson 6.3 – Writing Better Code . . . .

Learning Objectives . . . .

How can I improve my code for computational efficiency? . . . .

How does hardcoding impact a program? . . . .

What are some tips for good comments and spacing? . . . .

Why does proper code indenting matter? . . . .

What are some tips for choosing inputs and outputs? . . . .

What are some tips for thinking ahead in when designing my program?

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 7: FUNCTIONS

Lesson 7.1 – User-Defined Functions . . . .

Learning Objectives . . . .

What is a function? . . . .

What are the naming rules for functions in MATLAB? . . . .

How can I create functions in MATLAB? . . . .

Can I define functions in the program m-file? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 7: FUNCTIONS

Lesson 7.2 – Function Design and Communication . . . .

Learning Objectives . . . .

How can I add a description for my function? . . . .

How can I define errors and warnings inside my function? . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 8: Loops

Lesson 8.1 – while Loops . . . .

Learning Objectives . . . .

What is a loop? . . . .

What is a while loop? . . . .

Lesson Summary . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 8: Loops

Lesson 8.2 – for Loops . . . .

Learning Objectives . . . .

What is a for loop? . . . .

How can I reference vectors inside of a loop? . . . .

Do I have to use the loop counter variable in the body of the loop? . .

When do I use a for loop vs. a while loop? . . . .

Lesson Summary . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 8: Loops

Lesson 8.3 – break and continue Commands . . . .

What are the `break` and `continue` commands? . . . .

How does the `break` command work in MATLAB? . . . .

How does the continue command work in MATLAB? . . . .

Lesson Summary . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 8: Loops

Lesson 8.4 – Nested Loops . . . .

Lesson Objectives . . . .

What is a nested loop? . . . .

How do nested loops work? . . . .

How do loop mechanics apply to nested loops? . . . .

What is a “flag”? . . . .

When should I use programming flags? . . . .

How can I use `break` and `continue` in nested loops? . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 8: Loops

Lesson 8.5 – Working with Matrices and Loops . . . .

Learning Objectives . . . .

How can I reference matrices in a loop? . . . .

How do I store values in a matrix using a loop? . . . .

How can I access specific areas of a matrix? . . . .

What is vectorization? . . . .

How can I vectorize matrix operations in MATLAB . . . .

What are some tips I can use for vectorization? . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 8: Loops

Lesson 8.6 – Applied Loops . . . .

Learning Objectives . . . .

Why is this lesson important? . . . .

How can I sort an array? . . . .

How can I find the sum of a vector? . . . .

How can I plot different variations of a function using loops? . . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 9: READING FROM AND WRITING TO FILES

Lesson 9.1 – Reading from Files . . . .

Learning Objectives . . . .

Why read data from a file? . . . .

How do I read numeric-only data from files? . . . .

What is a delimiter? . . . .

How can I read numeric and character data from files? . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 9: READING FROM AND WRITING TO FILES

Lesson 9.2 – Writing to Files . . . .

Lesson Objectives . . . .

How can I write numeric data to files with MATLAB? . . . .

How can I write non-numeric data to files with MATLAB? . . . . . .

Lesson Summary of New Syntax and Programming Tools . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Module 9: READING FROM AND WRITING TO FILES

Lesson 9.3 – Navigating Directories in MATLAB . . . .

Lesson Objectives . . . .

How do I set the current working directory for MATLAB? . . . .

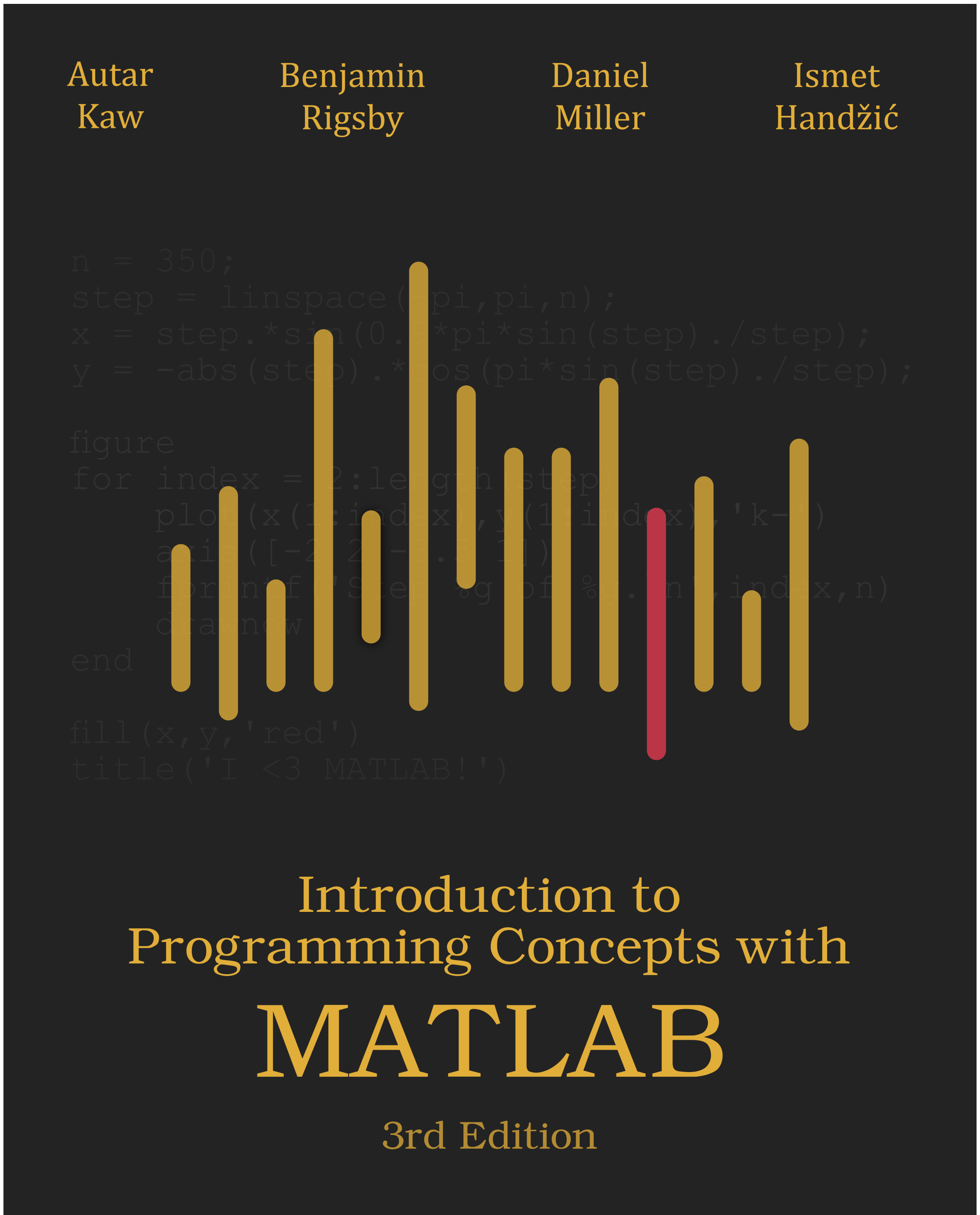
How can I loop through the contents of a directory? . . . .

How can I create a new folder (directory) with MATLAB? . . . .

Multiple Choice Quiz . . . .

Problem Set . . . .

Front Cover



Autar Kaw Benjamin Rigsby Daniel Miller Ismet Handžić

Introduction to Programming Concepts with MATLAB

3rd Edition

Source: Designed by Ismet Handžić and Benjamin Rigsby

Information

Authors: Autar Kaw, Benajmin Rigsby, Daniel Miller, Ismet Handzic

License

Dedication

To Sherrie, Candace, Angelie, and Bucky J Barks (AK)

To Victoria (BR)

To my mother, Bonny Miller (1958 – 2015) (DM)

To my family (IH)

Thoughts

“More than just teach you how to program, this course teaches you how to think more methodically and how to solve problems more effectively. As such, its lessons are applicable well beyond the boundaries of computer science itself. That the course does teach you how to program, though, is perhaps its most empowering return. With this skill comes the ability to solve real-world problems in ways and at speeds beyond the abilities of most humans.” - *David Malan, who teaches a general computer science course at Harvard to majors and non-majors of computer science.*

“Writing computer programs to solve complicated engineering problems and to control mechanical devices is a basic skill all engineers must master” – *Harry Cheng who teaches computer programming to mechanical engineering undergraduates at the University of California, Davis.*

Author Bios

AUTAR KAW

Autar Kaw is a professor of mechanical engineering at the University of South Florida. He is a recipient of the 2012 U.S. Professor of the Year Award from the Council for Advancement and Support of Education and Carnegie Foundation for Advancement of Teaching.

Professor Kaw's primary scholarly interests are in education research methods, open courseware development, flipped and adaptive learning, bascule bridge design, fracture mechanics, composite materials, and the state and future of higher education.

Funded by National Science Foundation (2002-23), under Professor Kaw's leadership, he and his colleagues from around the nation have developed, implemented, refined, and assessed online resources for open courseware in Numerical Methods (http://nm.MathForCollege.com). This courseware annually receives 1,000,000+page views (http://mathforcollege.com), 2,000,000+ views of the YouTube lectures (http://youtube.com/numericalmethodsguy), and 90,000+ visitors to the "numerical methods guy" blog (http://AutarKaw.org). This body of work is also used in the understanding of the impact of the flipped, blended and adaptive settings on cognitive and affective learning gains of engineering students.

Professor Kaw has written more than 100 refereed technical papers and his opinion editorials have appeared in the Tampa Bay Times, Tampa Tribune and Chronicle Vitae. His work has been covered/cited/quoted in many media outlets, including the Chronicle of Higher Education, Inside Higher Education, U.S. Congressional Record, Florida Senate Resolution, ASEE Prism, Times of India, NSF Discovery, and Voice of America.

Web: http://AutarKaw.com

YouTube: http://youtube.com/numericalmethodsguy;

Blog: http://blog.AutarKaw.com;

Twitter: http://www.twitter.com/numericalguy

BENJAMIN RIGSBY

Benjamin Rigsby is a Ph.D. candidate in mechanical engineering at the University of South Florida (USF) in Tampa, Florida. He received his Bachelor of Science degree in 2015 and Master of Science degree in 2017: both in mechanical engineering from USF. After graduation, Ben plans to work in the industry as an engineer in research and development.

Ben has been the instructor twice for the Programming Concepts for Mechanical Engineers course at USF. He has worked as a teaching assistant since 2014 in several mechanical engineering courses while developing course materials and assisting students. Ben also works as a research assistant at USF in the Rehabilitation Engineering and Electromechanical Design lab under the guidance of his advisor Professor Kyle Reed. Ben's research focuses on the areas of human-robot interaction, force perception, and haptics.

In his spare time, Ben enjoys 3D printing, gaming, traveling, and making educational online content. Contact information as well as further information on his current research, teaching, and professional information can be found at benjaminrigsby.com.

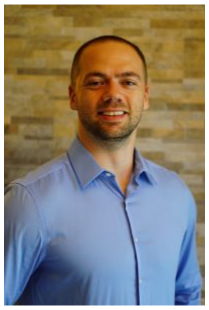
DANIEL MILLER

Daniel Miller is an alumnus of the University of South Florida. He majored in mechanical engineering and received a B.S. degree in 2009, followed by an M.S. degree in 2011. As a graduate student, he first worked as a teaching assistant for the Programming Concepts for Mechanical Engineers course and then instructed the class from 2010 to 2011. He received the USF Provost's Graduate Teaching Assistant Award in 2011.

Dan also worked as a research assistant in the field of numerical methods related to the design and analysis of body armor systems using computational methods. Dan is a registered Professional Engineer in Florida and a certified Project Management Professional with the Project Management Institute. He is a co-inventor of hybrid wearable body armor and was awarded a U.S. Patent for the system in 2013.

Dan is currently employed as a mechanical design engineer in a multinational company in Tampa FL and is a member of the U.S. Navy Reserve.

ISMET HANDŽIĆ



Ismet Handžić completed his Bachelor of Science degree in Mechanical Engineering at Western Kentucky University in 2009. Handžić continued his education for a Master of Science degree in Mechanical Engineering at the University of South Florida. After completion in 2011, he continued to pursue his Ph.D. degree in Mechanical Engineering. The general topics included in his doctorate dissertation involved walking rehabilitation, rolling dynamics, passive synchronization and dynamics, string vibration, and computer simulation of walking patterns.

Handžić concluded his graduate work with twenty peer-reviewed publications and five utility patents. During his time in graduate school, Handžić enjoyed being a graduate teaching assistant, actively trying to find original ways to create effective teaching materials. Subsequent to his studies, Handžić joined a small startup company to develop three of his patented and licensed inventions. These inventions included the Moterum M-Tip crutch tip and the Moterum iStride stroke rehabilitation shoe.

Handžić's successive positions in the industry included a mechanical research engineer at a crossbow weapon manufacturer, analyzing and designing crossbow components, and a system test engineer at an aerospace company designing and programming electromechanical automated test equipment for electrical components. His current employment is as a system test engineer at an IoT technology start-up company, developing automated test equipment and programming various automated tests of IoT devices.

In his leisure time, Handžić likes to spend time with his wife and kids, tinkering on small maker projects, programming, photography, playing his guitar, or hammering on larger projects such as the contents of this textbook.

Preface

This book is intended for an introductory course in programming in STEM (science, technology, engineering, and mathematics) fields while using MATLAB as the programming language. MATLAB is a popular computational software package used in universities and industries alike.
This textbook differentiates itself from others in two specific ways.

1) The textbook is suitable for the many engineering departments throughout the nation that no longer teach a 3-credit hour programming course. They weave programming and mathematical software packages such as MATLAB in courses such as Foundations of Engineering, Freshmen Design, Modeling of Systems, Engineering Analysis, Numerical Methods, etc. This book is highly suitable for such audiences. To achieve these goals and make the access far-reaching, we have been deliberate in keeping the lessons short in length so that instructors can easily choose the course content in a modular way.

2) The textbook is a stand-alone resource for learning programming where the lectures complement the textbook rather than vice versa. This is because of the reason above where in-classroom time is truncated, and therefore students need to be more self-taught. For this reason, we have been meticulous when selecting and organizing the textbook content to include fundamental and application programming problems that prepare students well for other problems they will solve in academia and industry.

The book has nine modules which have been each broken down by lessons. There are 42 lessons in all and depending on the learning outcomes of the course, an instructor can choose to assign only necessary lessons. Modules 1-3 focus on MATLAB and programming basics like the MATLAB program interface, programming variables, different types of data, debugging, plotting, and applications to science and engineering problems. In Module 4, we show the use of MATLAB for basic mathematical procedures learned in the engineering courses including nonlinear equations, integration, differentiation, simultaneous linear equations, interpolation, regression, and ordinary differential equations.

In Modules 5-8, the user is introduced to basic programming concepts of conditional statements, repetition (loops), and custom functions. In Module 9, program input/output is shown with writing to and reading from external files as well as navigating directories with MATLAB. Important appendices include a primer on matrix algebra, a collection of mini-projects, and a introduction to animating plots in MATLAB. Appendix A provides a primer on matrix algebra. Appendix B contains a set of mini-projects. Appendix C demonstrates how to make animated plots in MATLAB.

Each lesson contains screenshots of actual MATLAB programs that are used to help illustrate the concepts presented. More than 120 complete programs are shown throughout this book to demonstrate to the reader how to use programming concepts. The book is written in a USA-Today style question-answer format for a quick grasp of the concepts.

The purpose of this book is to provide the reader with a firm basic understanding of MATLAB syntax and fundamental programming concepts. Each lesson contains MATLAB programs that are used to help illustrate the concepts presented. By no means do the authors claim to present every MATLAB command, function, application, or programming concept in existence.

Contact Information

We would appreciate feedback, questions, or comments that you may have on this book. We are especially looking for any typographical errors. We will update these immediately with the publisher and also we will keep a complete list of corrections at programming.autarkaw.com/errata.html.

You can contact the first author, Autar Kaw, via: Email: AutarKaw@yahoo.com Telephone: +1 (813) 974-5626 Twitter: numericalguy Mailing Address: Department of Mechanical Engineering, ENG030 University of South Florida 4202 East Fowler Avenue ENG030, Tampa, FL 33620-5350

Acknowledgements

Kaw would like to thank his spouse, Sherrie, and children Candace and Angelie, who encouraged him to first co-write this textbook with Miller, and now with two more co-authors, Rigsby and Handžić. Miller would like to thank his spouse, Lisa.

What is New With This Edition

We have rethought the layout of the book by grouping sets of lessons into modules that address a specific set of fundamental topics as well as reordered some lessons for learning clarity.

We have added ten new lessons and extended other lessons to more fully cover programming fundamentals. Additionally, there are more than 50 new MATLAB example codes.

Based on student feedback over the last seven years, we have reformatted the whole book for readability and clarity.

We have added end-of-lesson summaries of the new syntax, functions, and commands covered in each lesson to make referencing and reviewing faster.

New figures have been added to visually demonstrate fundamental concepts.

We have updated all syntax and example codes to reflect MATLAB, that is, R2018b.

A Note to Students

What will I be able to do after completing this book?
Imagine you are given a file that contains data of position and time of the path of a rocket. By the end of this book, you will be able to read in the data from the file(s), estimate position, velocity, and acceleration of the rocket, and plot each of these dynamics variables simply by clicking “run” on your program. The best part is, if you get new data or multiple sets of data from multiple rockets, you can get all of these results again with only minimal additional work. This is just one example of many real-world problems you will be capable of solving after mastering the material in this book.

This textbook will give you a strong foundation in programming fundamentals through MATLAB. Although some more advanced topics like object-oriented programming are beyond the scope of this book, you will be able to solve the vast majority of engineering problems you encounter in school and in the workplace using the knowledge and skills you gain from this book.

Why should I learn programming? It may be a common belief that the concepts learned in programming are only applicable to computers and computer languages. However, this is not true. The various concepts of programming, for example, a yes/no decision, are used in nearly every action we take while interacting with the world in our daily activities. For instance, you may ask yourself whether you should drink tea before going to sleep, or whether you should exercise before eating a meal.

The typical sequential structure of a computer program is also used by us as we order the events of our schedule to make sense. For instance, one would never consider putting on their shoes before their socks. Logically, an individual will first put on their socks, then their shoes and finally, they would secure the shoes.

David Malan who teaches a general computer science course CS50 at Harvard to majors and non-majors of computer science (largest course at his institution and the largest Massive Open Online Course (MOOC) on edX) sums it up the best - “More than just teach you how to program, this course teaches you how to think more methodically and how to solve problems more effectively. As such, its lessons are applicable well beyond the boundaries of computer science itself. That the course does teach you how to program, though, is perhaps its most empowering return. With this skill comes the ability to solve real-world problems in ways and at speeds beyond the abilities of most humans.”

Furthermore, programming will teach you important debugging skills that are useful in correcting all sorts of mechanical and electrical systems. You will learn the steps to identify a problem, determine its cause, and finally devise a solution. You will learn to be meticulous when comparing what you expect with what you observe.

How can I use the book most effectively? After reading each lesson, do all of the multiple-choice questions and as many exercise problems as you can (preferably all). Practice is essential when learning to program. Cramming will not work, and as with many other courses, repeated bursts of practice is the best method to grasp the material (an hour or two each day). When completing the exercises, it is highly recommended to work alone as the approach to new problems needs to be learned individually. This will make your debugging and testing skills much stronger, which will, in turn, make you a better programmer.

Pay careful attention to the “Important Notes” in the text. We have been deliberate about placing these in the lessons so that they can be helpful without being overwhelming. They are meant to be a kind of pro-tip to tell you something that some people only realize after making the mistake many times.

Also take advantage of the Index of terms, functions, and commands at the back of the book. This can be a quick way to find that one function you need to review.

How does MATLAB compare to other popular languages? MATLAB is a powerful programming language with many first-party functions and commands to do all kinds of tasks like statistics, machine learning, controls, data analysis, modeling, and user interfaces to name a few. It also has excellent documentation compared to other popular languages due, in part, to the fact that MATLAB is a proprietary language.

Learning MATLAB will give you a great foundation to transfer to other languages should you need to. Python is a popular open-source programming language that has similar syntax compared to MATLAB. Suffice it to say, MATLAB is a good choice as a first language both for its ubiquity in academia and for its stellar documentation (make sure you take advantage of this!).

A Note to Instructors

Scope of Textbook We have endeavored to include all of the necessary fundamental programming syntax and skills for a student to solve most problems they will encounter in STEM. We aimed to not only teach MATLAB syntax in this book, but inform and inspire good programming, documentation, debugging, and program planning and research practices. We believe that this will prepare students well for tackling new MATLAB functionality and building on the programming knowledge they gain from this book. A brief summary of the objectives of each module is given below.

Module 1 introduces how to interact with the MATLAB program including opening and saving m-files and its basic components like the Editor and Command Windows.

Module 2 introduces basic programming fundamentals including the concept of a variable, different data types like numbers and strings, and numeric arrays as used in the context of mathematics and MATLAB. The reader is introduced to program design with user inputs and program outputs and is encouraged to think about the beginning and the end of a program rather than just direct solutions to a problem.

Module 3 introduces how to visualize different types of data in MATLAB, which includes how to plot discrete data pairs directly as well as from discrete data generated from continuous functions. Advanced visualizations in MATLAB are covered including bar graphs and polar and 3D plots. The essential MATLAB plot properties that accompany these plots are demonstrated.

Module 4 introduces how to use MATLAB functions to conduct differentiation and integration, curve fit via interpolation and regression, solve for roots of nonlinear equations, find solutions to simultaneous linear equations, and solve ordinary differential equations.

Module 5 introduces conditions and conditional statements including the relevant Boolean logic.

Module 6 introduces tools for program design and communication including pseudocode and flowcharts. Tips for program design and communication are also provided.

Module 7 introduces user-defined functions where readers are shown how to write their own custom functions. Tips on how to consider the user of a function are also given.

Module 8 introduces loops and provides thorough coverage of the topic. Many different cases are covered including use of matrices and loops together and the obligatory summing, searching and sorting. Other examples include implementing recursive formulas and approximating mathematical functions using infinite series.

Module 9 introduces interacting with external files and directories. Methods for reading from and writing to text and Excel files are given. Applications demonstrating how to interact with data, once it has been read into MATLAB, are also provided.

Appendix A provides a primer on linear algebra which describes fundamental matrix operations used in programming such as addition, multiplication, inverse, and many more. Special types of matrices, such as symmetric, diagonally dominant, identity and several more are also defined.

Appendix B contains a set of mini-projects that thoughtfully provide additional practice to the student. Relevant modules are noted at the beginning of each mini-project for easy reference.

Appendix C demonstrates how to animate 2D and 3D plots and data in MATLAB.

Tips on Using the Book for Instructors

For this edition, the textbook was intentionally rearranged into nine modules with a total of 42 lessons. The textbook will appeal to schools ranging from where programming is introduced to freshmen in a first-year engineering design course to those who have a full-fledged 3-credit hour course dedicated to programming at a higher level. The intention is that the instructor would choose the lessons that are appropriate in each module for their students based on the course level and effort. Our recommendation for courses, such as Numerical Methods with Programming or Engineering Analysis, where programming is instead introduced as one of several topics, would be to safely skip the following lessons: Lessons 3.3, 4.3 to 4.9, 8.6, 9.1 to 9.3.

At the end of most lessons, there is a multiple-choice question quiz and a set of exercises. You should encourage students to finish both problem sets. The course works well by assigning a set of mini-projects deliverable every other week, and these have been included in the end of lesson exercises as well as in Appendix B.

Students Program Submissions We have included instructions on publishing m-files in Lesson 1.7. We have found it very helpful for students to include a published version of their program. This is for three main reasons: 1) it reduces the number of m-files that need to be run while grading, 2) the outputs are immediately shown after the appropriate code, which is helpful to both the grader and the student, and 3) it encourages students to review the output of each submitted problem.

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.1 – MATLAB Introduction

Learning Objectives

After reading this lesson, you should be able to:

1) *know why MATLAB is useful for engineers and scientists,*
2) *get MATLAB on your computer,*
3) *comprehend the free alternatives to MATLAB.*

What is MATLAB?

MATLAB is a high-level programming language built for scientists and engineers who work in academia and industry. MATLAB is relatively user-friendly and provides many out-of-the-box tools and resources relevant to engineers for mathematics, controls, data analysis, data manipulation, data acquisition, and more (see MathWorks documentation for a full list of toolboxes).

This software has built-in abilities like plotting and statistical analysis that many other popular programming languages do not have, which makes it attractive to engineers who are more focused on the problem-solving rather than the computer science happening behind the scenes. For example, MATLAB offers a “mean” function to compute the average of a set of data. The user can simply use this function without specifically adding data values and then dividing by the number of data values.

What is MATLAB used for in engineering and science?

While MATLAB is capable of all of these and more, this topic will only cover the essential utilities, while presenting practical examples. These essentials can be further used to figure out the more advanced utilities.

- *Data manipulation* - Math operations, statistics, linear algebra, vector and matrix manipulation, optimization, etc.
- *Data Presentation* - Plotting/graphing (2D/3D), animation (2D/3D), graphical user interface (GUI) application, remote deployment, etc.
- *Hardware Communication and Control* - Sensor reading, actuator control, signal processing, signal generation, etc.
- *Media Processing* - Computer/Machine Vision, audio recording/playback, frequency analysis, etc.
- *And much, much more...*

How can I get MATLAB onto my computer?

MATLAB is a proprietary programming language held by its parent company MathWorks, which means you, your school, or your company needs to pay for a license to use it. You can see buying options and pricing at MathWorks, but they do offer a student version.

If you do buy MATLAB and have the option to include toolboxes for a discounted price, you should consider doing so. A student version can be downloaded for as low as $49 (as of November 2019). MATLAB is highly modular, meaning that various specialized toolboxes can be added to the core software. Some toolboxes come standard with MATLAB, while others have to be purchased. Some of the toolboxes, such as the Symbolic Math Toolbox, are fundamental to common uses like solving equations.

To install MATLAB, follow the usual procedure for your operating system. There is no need to set up environmental variables, set a path to a compiler, etc. (if you have done that for other programming languages). If you run into trouble during the installation process, go to https://www.mathworks.com/help/install/index.html for details on how to install your version of MATLAB.

Are there any free alternatives to MATLAB?

Not every engineer or engineering company is able to afford a software license to use MATLAB. However, many universities and large companies provide MATLAB licenses to students/employees free of charge, so check with your university/company first! The list below shows the current most popular numerical analysis software alternatives to MATLAB. In most cases, these alternatives offer nearly identical commands and syntax/code structure.

1. Octave Online: fastest to use, no setup/install required
2. Octave: most common alternative, the closest program to MATLAB
3. SciLab and FreeMat.

Where can I find more information and help with MATLAB online?

Complete documentation and examples on MATLAB commands and functions can be found at MathWorks.com. We will go over how to use documentation in later lessons. A good and complete MATLAB overview, relating to programming and programming languages in general, can be found on Wikipedia.

Multiple Choice Quiz

The content of this page is intentionally blank

Problem Set

The content of this page is intentionally blank

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.2 – Hello World

Learning Objectives

After reading this lesson, you should be able to:

1) *perform the basic steps to start a new program in MATLAB,*
2) *run a simple program.*

This lesson walks you through creating the classic "Hello World" program. Unlike other lessons, we will not explain much of the "how" or "why" in this lesson. *You do not need to understand everything in this lesson: just observe!* The purpose here is to give beginning programmers something to relate to in subsequent lessons.

Where can I find and open the MATLAB program?

The procedure for opening MATLAB® is described in this section for Windows. First, go to the start menu and look for "MATLAB" in the list of your installed programs. Once there, click on the appropriate version of MATLAB installed on your computer. Do not click on the "Activate/Deactivate" options if MATLAB is already installed on your computer. When the program opens, a window that is similar to the one in Figure 2 should open.

Step 1: Create a New m-file

First, we need to create a file to store our program. In the top left portion of the MATLAB screen, create a new m-file by pressing the "New Script". Note that in many languages a script is a common term for the file that holds your code. Save the blank file as "helloWorld".

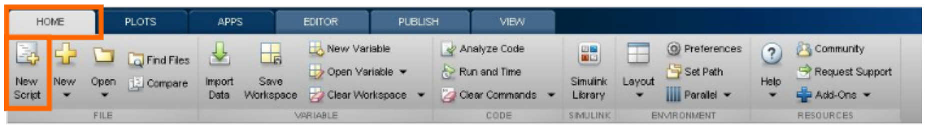


Figure 1: Click "New Script" on the far left under the "Home" tab to open a new m-file.

Step 2: Write the 'Hello World' Code

In this next step, we will write the code that will output/return our message: '`Hello World`'. Note that the green text after `%` will be ignored and will not be executed. See Figure 2 if you are not sure where to type your code.

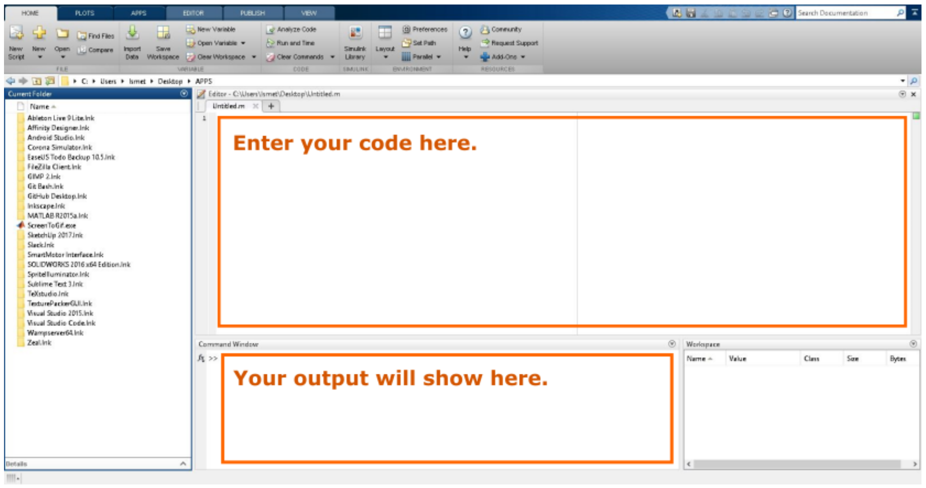
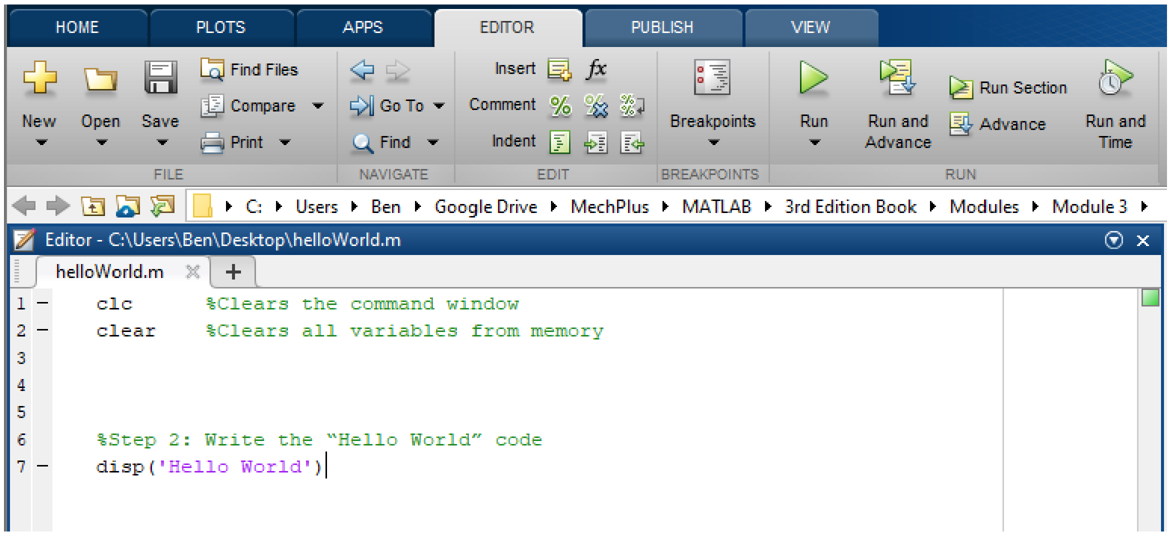


Figure 2: MATLAB window showing where to type your code (in Editor window).

Example 1

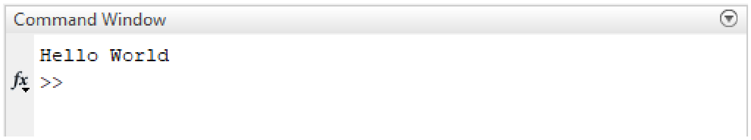
Write a program that outputs the message “Hello World” to the user.

Solution



```
clc       %Clears the command window
clear     %Clears all variables from memory



%Step 2: Write the “Hello World” code
disp('Hello World')
```

Figure 3: Output a message to the user.



```
Hello World
>>
```

Figure 4: The Command Window output for Example 1.

Step 3: Run the Program

To get our program to do something, we need to “run” the program. Figure 5 shows the “Run” button, which you should click to run the program. This is located under the “Editor” tab in the toolbar (top left of screen).

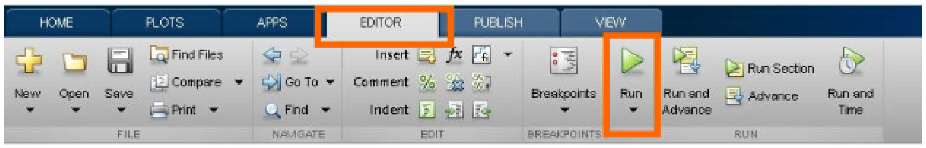


Figure 5: Run the program by clicking the highlighted button under the "Editor" ribbon tab (top).

Once the "Run" button is pressed, the program will execute our command. The result of our code will show in the Command Window (the window on the right).

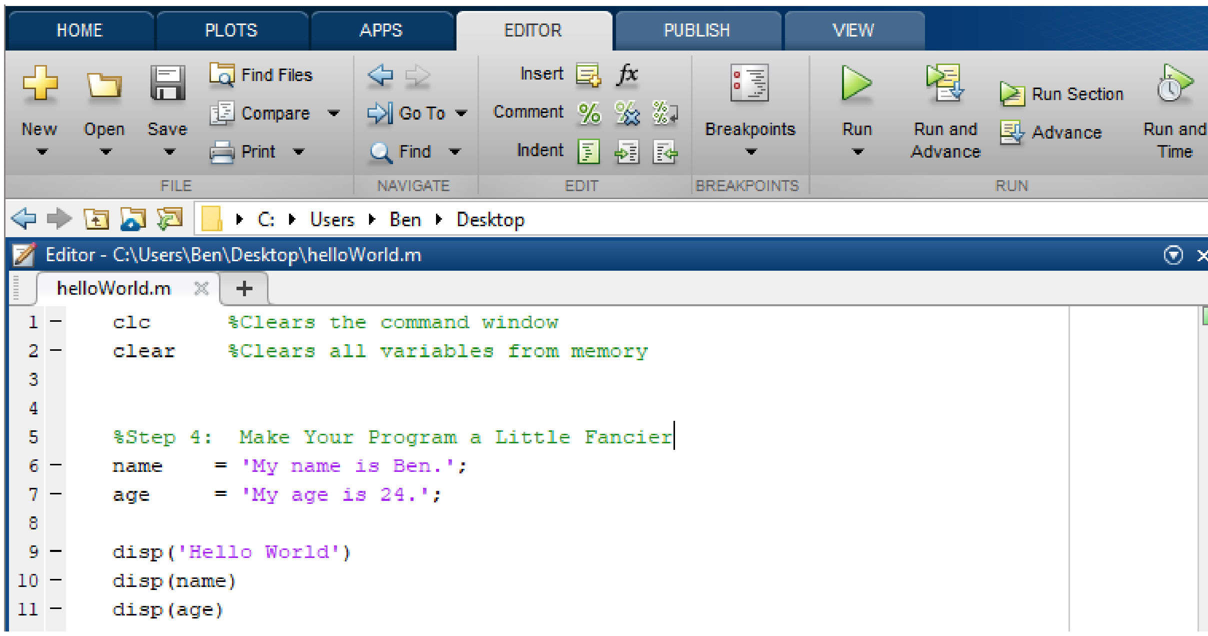
Step 4: Make Your Program a Little Fancier

Finally, we can do a couple more things to demonstrate some other simple concepts. This time we will include our name and age in the "Hello World" greeting. Again, just observe. Each of these concepts will be explained in the lessons to come.

Example 2

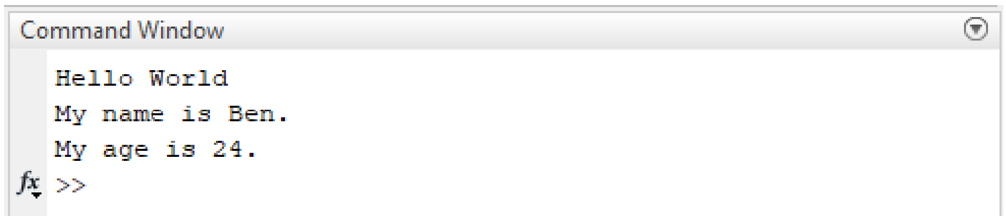
Write a program that outputs a fancy "Hello World" to the user including your name and age.

Solution



```
clc      %Clears the command window
clear    %Clears all variables from memory


%Step 4:  Make Your Program a Little Fancier
name     = 'My name is Ben.';
age      = 'My age is 24.';

disp('Hello World')
disp(name)
disp(age)
```

Figure 6: Output a fancier message to the user.



Command Window

```
Hello World
My name is Ben.
My age is 24.
>>
```

Figure 7: The Command Window output for Example 2.

Multiple Choice Quiz

The content of this page is intentionally blank

Problem Set

The content of this page is intentionally blank

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.3 – MATLAB Environment

Learning Objectives

After reading this lesson, you should be able to:

1) *know the main components of the MATLAB environment,*
2) *use the MATLAB software to write code,*
3) *use the interface to find and set the current working location of MATLAB.*

MATLAB Environment Windows and Parts

The MATLAB Environment is all the windows, sub-windows, and parts that are part of the MATLAB program (note, both the software interface and the actual computer language are called MATLAB). By default, the components of the MATLAB Environment are laid out as shown in Figure 1. Note that we are referencing MATLAB versions R2015a and above in this lesson. Previous versions have some similarities, but R2012a and before are significantly different visually (no ribbon navigation). However, the basic parts like the Editor and Command Window are fundamentally the same throughout the different versions.

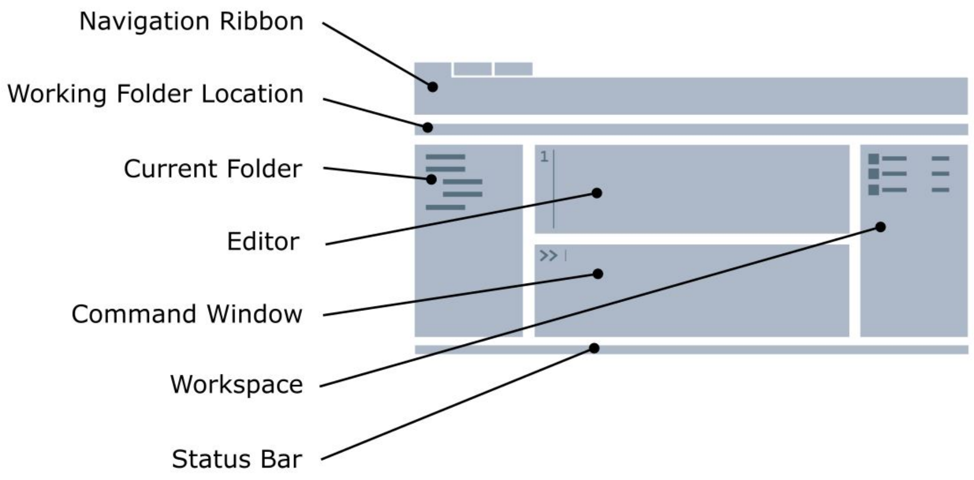


Figure 1: The major windows and section of the default setup of the MATLAB Environment.

The following sections of this lesson will review the individual and major parts of the MATLAB Environment seen above in Figure 1. You can refer back to Figure 1 and/or open MATLAB to see physically where the different pieces are as you progress through this lesson. It is essential for a beginner to understand each component and its function to write MATLAB code.

Navigation Ribbon

As with most modern programs the navigation ribbon is a way to help users find, understand, and use MATLAB software features efficiently and directly. The navigation ribbon has different tabs that lump similar software functionality together. Within a ribbon tab, you can find different groups. An example of this can be found in the "Home" tab that includes the "File" group, which includes various functions and options for working with MATLAB files such as "New", "Open", or "Compare".

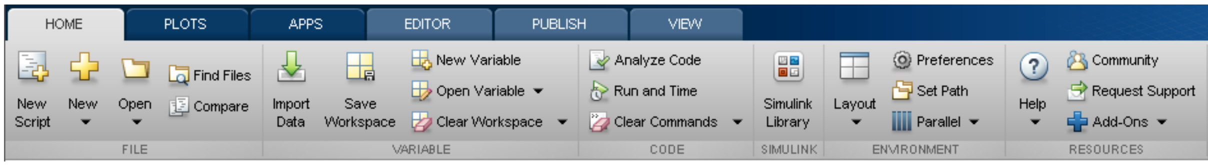


Figure 2: The navigation ribbon in the MATLAB Environment.

The most common location on the ribbon that you will use is "Editor" tab. Under the Editor tab, you can create a new m-file (a file for your code) and run that file. You can also see the “Save” button in the Editor tab, which will save your m-file. Note that running the m-file automatically saves your m-file. One way or the other, it is a good idea to save your files often.

The “*Publish*” tab contains functions that can be used to present your code and MATLAB work in a presentable way. This functionality of the “Publish” ribbon tab can be reviewed in Lesson 1.7.

The “*Plots*” and “*Apps*” tabs contain modules and functions that are shortcuts to a specific piece of MATLAB code. Although the shortcuts can be convenient and efficient to a particular workflow, they are not essential to this course: we are just mentioning them informatively here. You may want to play around with these ribbon tabs and explore their functionality later in the course.

Working Folder Location

The working folder location bar shows the current directory that MATLAB is referencing (see Figure 3). This is the computer folder from which your code will run. If the working directory is different than the location of your code, MATLAB will prompt to change the working folder location (shown in Figure 4). The working folder location bar works in conjunction with the current folder window, which is explained in the following section.

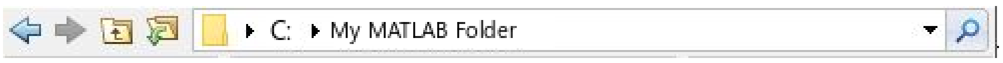
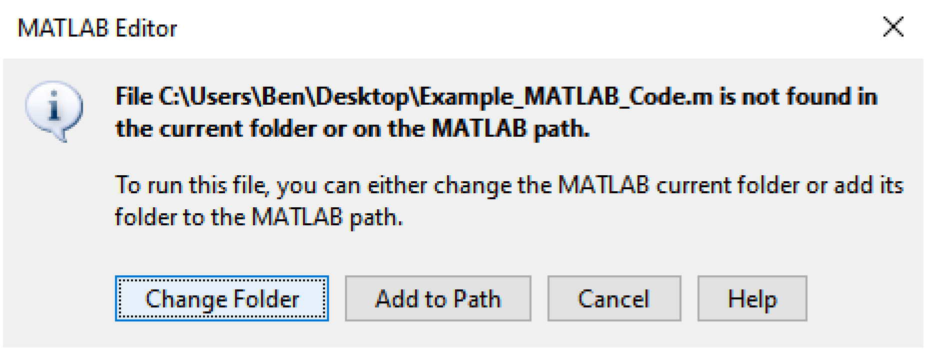


Figure 3: The working folder display in MATLAB.

Figure 4 shows the prompt you will get if the m-file you are trying to run is not in the MATLAB current folder or on one of its (directory) paths. “Change Folder” means: make the folder that contains this m-file the “current folder”. “Add to Path” means add the folder location of the m-file you are trying to run to the MATLAB paths. A path is a more permanent directory that MATLAB will always look in when trying to find a program or function you are running/calling. Simply put, choose “Change Folder” most of the time/if you do not know. Neither option will “break” anything, and both should work fine for our purposes.


MATLAB Editor

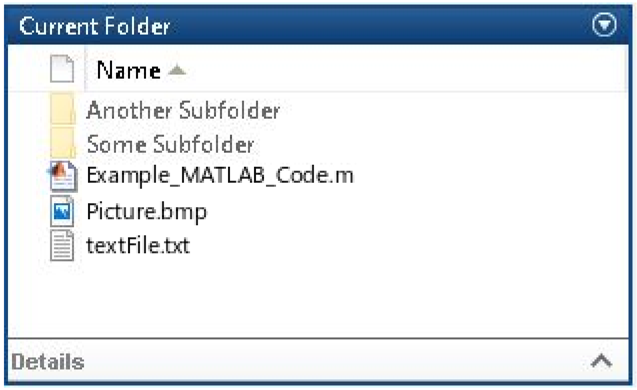
File C:\Users\Ben\Desktop\Example_MATLAB_Code.m is not found in the current folder or on the MATLAB path.

To run this file, you can either change the MATLAB current folder or add its folder to the MATLAB path.

Figure 4: MATLAB user prompt to change the current working folder.

Current Folder

The current folder MATLAB Environment sub-window, shown by default on the right-hand side of the window, shows you what files are in your working folder location (see Figure 5). This includes all subfolders and any type of files that are inside the working directory.



Another Subfolder
Some Subfolder
Example_MATLAB_Code.m
Picture.bmp
textFile.txt

Figure 5: Current folder contents window in MATLAB.

Command Window

The Command Window has a number of different uses in MATLAB. Generally, it acts as an output window and a temporary/quick coding environment. We will cover this window in depth in Lesson 1.6.

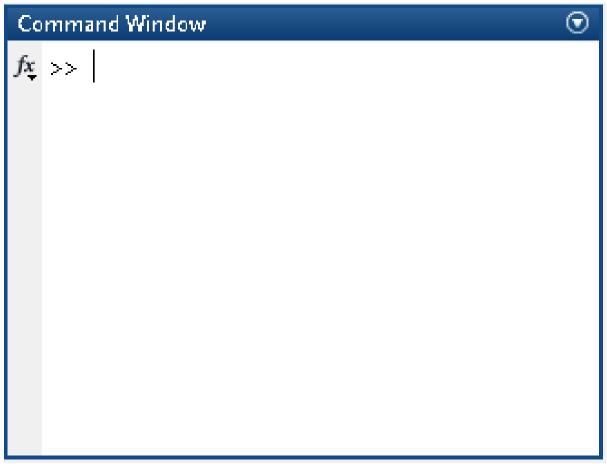
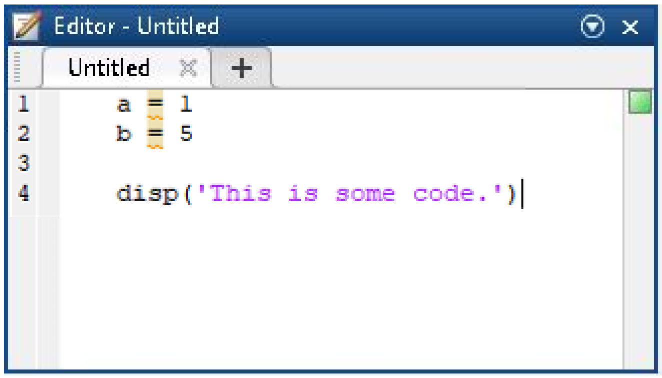


Figure 6: MATLAB Command Window.

Editor Window

While the Command Window can act as a temporary coding environment, the MATLAB Editor window is used to write outlines of code, or a script, that can be executed all at the same time. You can write an entire code sequence before running it. We will cover this window more in-depth in the following lesson on the m-file (Lesson 1.5). This window should not be confused with the Editor tab. They are related, but not the same thing as you can see in MATLAB.

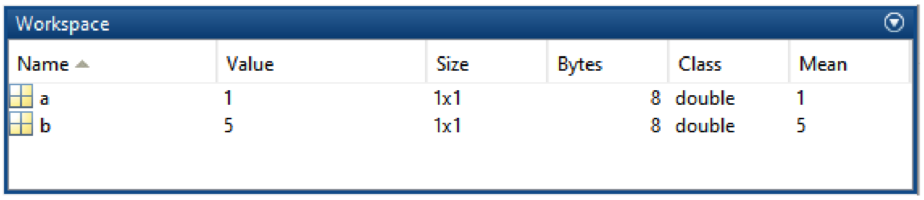


```
a = 1
b = 5

disp('This is some code.')
```

Figure 7: The Editor window (m-file editor) used to write and save a complete code.

Workspace

While programming, you will actively define variables to store data and values for later use. Such memory storage can range from storing text and numbers to storing large vectors and matrices. The workspace window (Figure 8) displays all variables that are currently defined and stored into memory. It can be helpful in that it shows different properties of that particular variable such as its type ("Class"), its matrix dimensions ("Size"), or the amount of memory used to store it ("Bytes"). You can even see quick facts about numerical data like the mean of the data.


Workspace

Name	Value	Size	Bytes	Class	Mean
a	1	1x1	8	double	1
b	5	1x1	8	double	5

Figure 8: The workspace window that is used to monitor any MATLAB data currently loaded into memory.

Status Bar

The status bar indicates the active status of the MATLAB program. Most of the time, there will be nothing important to display here. However, one particularly useful message is when MATLAB tells you it is "Busy" or "Waiting for an input". "Busy" generally means it is working on computations in the background from the last code you asked it to run. "Waiting for an input" refers to MATLAB waiting for a user input that you specified in your program (more on this in Lesson 2.4: Inputs and Outputs).

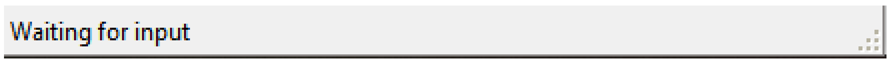

Waiting for input

Figure 9: The status bar that notifies the user of important MATLAB status messages.

Multiple Choice Quiz

The content of this page is intentionally blank

Problem Set

The content of this page is intentionally blank

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.4 – Changing MATLAB Preferences

Learning Objectives

After reading this lesson, you should be able to:

1) *choose different window layouts in MATLAB,*
2) *change the MATLAB user preferences.*

Most of the MATLAB preferences covered in this lesson are just that. However, there are a few changes that can make a functional difference while you are learning to program, so it would be wise to familiarize yourself with the options available.

How can I change the window layout in MATLAB?

MATLAB makes it simple to change the position and layout in the MATLAB program, and there are various ways to accomplish this.

One way is to use the layout options for each sub-window, which can be minimized, maximized, undocked, or closed by selecting the symbol in its upper right-hand corner as shown in Figure 1. Note that undocking means that the window can be separately and independently moved around from the MATLAB software (try it out!).

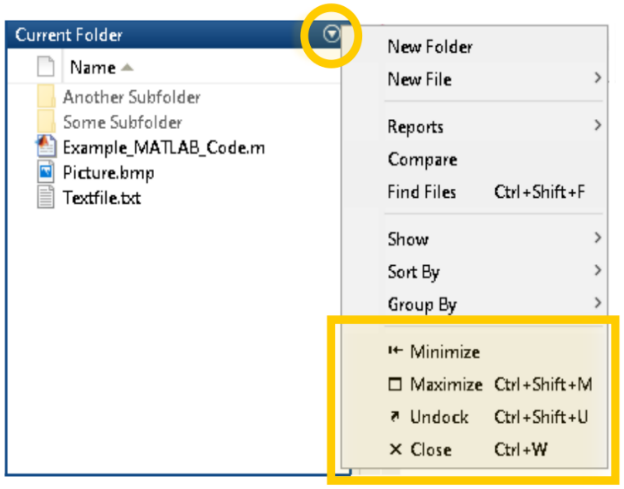


Figure 1: Changing view of a window in MATLAB.

If you happen to "lose" or accidentally close a window, the simplest way to recover it is by selecting one of the MATLAB preset layout options that can be found in the "Home" navigation ribbon and then under the "Layout" button as shown in Figure 2.

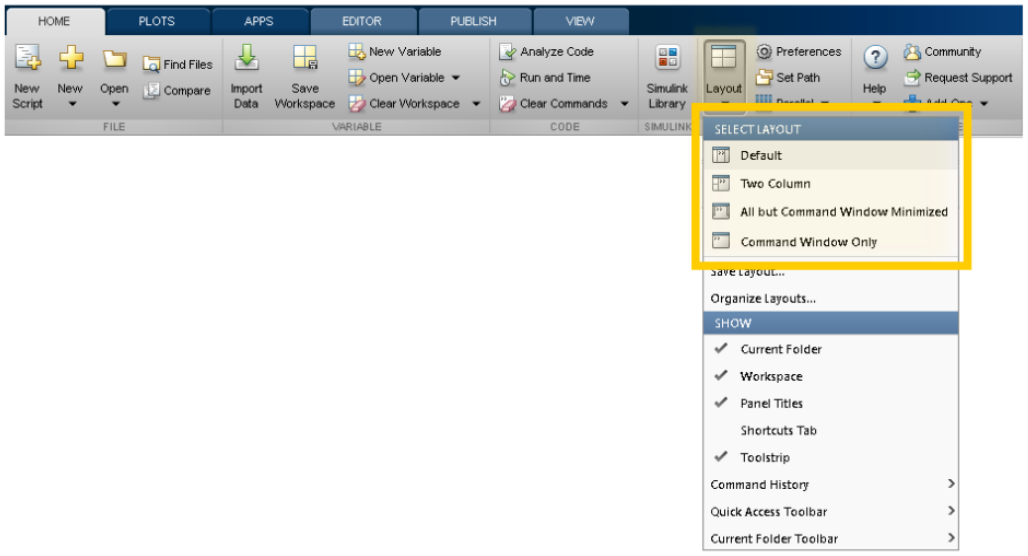


Figure 2: Selecting a layout in MATLAB.

✓ ***Important Note:*** Moving forward, this course will assume that you are working with the "Default" layout preset.

Changing Basic User Preferences

It is advantageous to change the default user preferences when working with MATLAB. For example, choose a more readable font or font size. These preferences may include visual/layout (font, font size) and functional (MATLAB behavior) options. While there are many fundamental preferences that can be changed, we will only demonstrate the font size here. Increasing the font size can make dense, small code feel more readable and less overwhelming.

Preferences can be found on the "Home" ribbon, in the "Environment" group, select the "Preferences" button as shown in Figure 3.

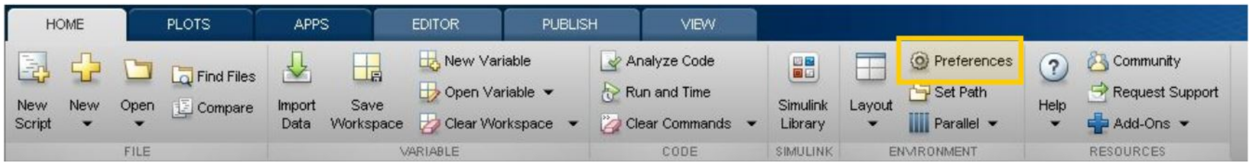
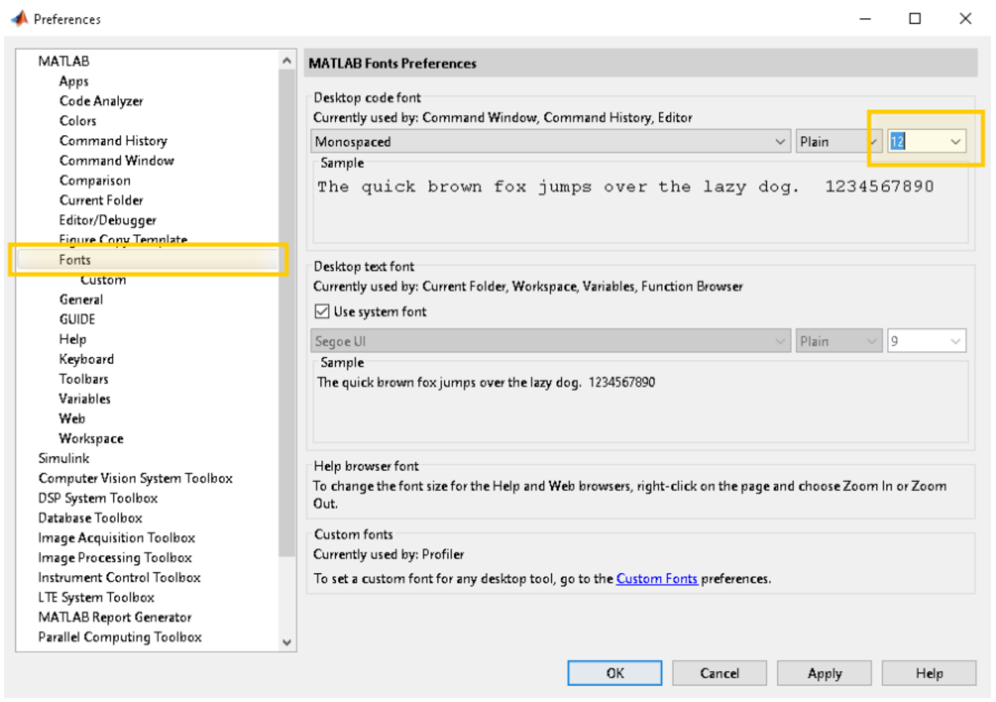


Figure 3: The MATLAB preferences location on the Home tab.

Once in the Preferences pop-up window (see Figure 4), select "Fonts" in the navigation tree on the right-hand side. Then in the "Desktop code font" section, select "12" as the font size. Once done, select "OK" at the bottom of the window.

Although it is not necessary to remember exactly how to change these preferences, it is good to keep in mind that most visual aspects of MATLAB are easily changeable to suit your needs/preference.



MATLAB Fonts Preferences

Desktop code font
Currently used by: Command Window, Command History, Editor
Monospaced Plain 12
Sample
```
The quick brown fox jumps over the lazy dog.  1234567890
```

Desktop text font
Currently used by: Current Folder, Workspace, Variables, Function Browser
☑ Use system font
Segoe UI Plain 9
Sample
The quick brown fox jumps over the lazy dog. 1234567890

Help browser font
To change the font size for the Help and Web browsers, right-click on the page and choose Zoom In or Zoom Out.

Custom fonts
Currently used by: Profiler
To set a custom font for any desktop tool, go to the Custom Fonts preferences.

Figure 4: MATLAB preferences window.

Multiple Choice Quiz

The content of this page is intentionally blank

Problem Set

The content of this page is intentionally blank

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.5 – The m-file

Learning Objectives

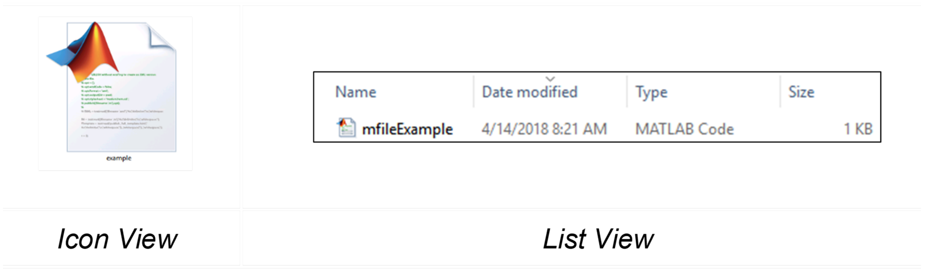
After reading this lesson, you should be able to:

1) *open and name a new m-file,*
2) *use the m-file to be able to solve mathematical problems,*
3) *change and manipulate data in the m-file,*
4) *use comments in an m-file.*

What is an m-file?

Just like how you may write a letter in a word processor program, one writes a MATLAB program in an m-file. This type of file is unique to MATLAB, but not much different than a text file which one could write or edit in text editors such as Notepad.

The m-file brings a whole new level of organization and ease of modifications to MATLAB that the Command Window cannot offer. The m-file and the Command Window are independent of each other but they communicate like best friends. The m-file is where all commands will be input, saved and then “run” through MATLAB, which evaluates the program and outputs the results in the Command Window. Changing any input, command, variable name or number in the m-file is easily done and does not require much time or effort. This is unlike the Command Window where if you have performed, for example, fifty statements, changing the first statement can be very difficult and time-consuming. The m-file can also be saved by the user to any directory, and reused or changed later. You can see examples of how this saved file (m-file) looks in the Windows File Explorer in Figure 1.



Name	Date modified	Type	Size
mfileExample	4/14/2018 8:21 AM	MATLAB Code	1 KB

Figure 1: This is what the m-file will look like in a file explorer (Windows in this case).

**Where can I create a new m-file?

First, open MATLAB. Now, to open a new m-file, click the “Home” tab in the top left corner of the MATLAB window (see Figure 2). Finally, click the “New Script” button. Note that this is the same procedure we went through in the Hello World lesson. A new window is now open in the MATLAB software; this is the m-file editor. You can change how these windows (Editor, Command Window, etc.) look in MATLAB, which we covered in Lesson 1.4.

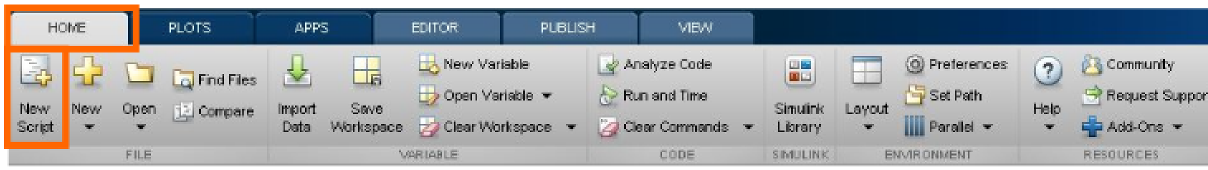


Figure 2: Click “New Script” on the far left under the “Home” tab to create a new m-file.

How do I save my m-file?

One of the most important things you can do in MATLAB is to save your m-file correctly. You can save your work by clicking the “Editor” tab, then selecting

"Save". You should make sure to save it under a name that makes sense to you. Do not keep the m-file name as the default "untitled1.m" as this can become very confusing later when you need to find the correct program. Follow the rules below for saving all m-files.

1. *Do not* start the m-file name with a number.
2. *Do not* place any spaces in the name of your m-file.
3. *Do not* use special characters in the name of your m-file such as *, /, +, $, %, &, #, @, etc. Underscores are permitted in m-file names.
4. *Do not* use any predefined commands to name the m-file.
5. *Do* name the m-file something that makes sense to you.

How do I input variables and expressions into the m-file?

MATLAB always reads the information in the m-file from top to bottom and from left to right (like a book). Also, each line of the m-file has the variable side and an expression side separated by an equal to sign. This is called an assignment and should not be confused with what we call an equation. If, for example, the lengths of two sides of a rectangle, `a` and `b`, are 2 inches and 6 inches, respectively, just type:

```
a = 2

b = 6
```

The data is now part of the m-file. Then, if we want to find the area of that rectangle, all we would need to do is type:

```
area = a*b
```

What we just did was type three lines of code, each line containing an assignment and altogether making a program. For MATLAB to run the program, and for the programmer to receive an output, the m-file must now be "run".

How do I run the m-file?

Running an m-file can be done by clicking on the “Editor” tab in MATLAB navigation ribbon, followed by clicking on “Run” (Figure 3). MATLAB will now execute the m-file and perform whatever code is specified, and then display the outputs in the Command Window. This is where the two independent windows communicate with each other; the outputs of the m-file are sent to the Command Window to be displayed. Figure 4 illustrates this for the example of finding the area of a rectangle.

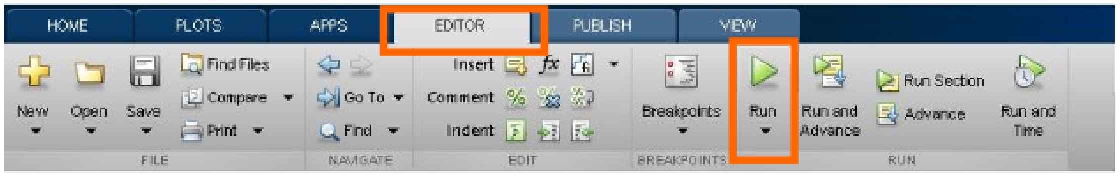
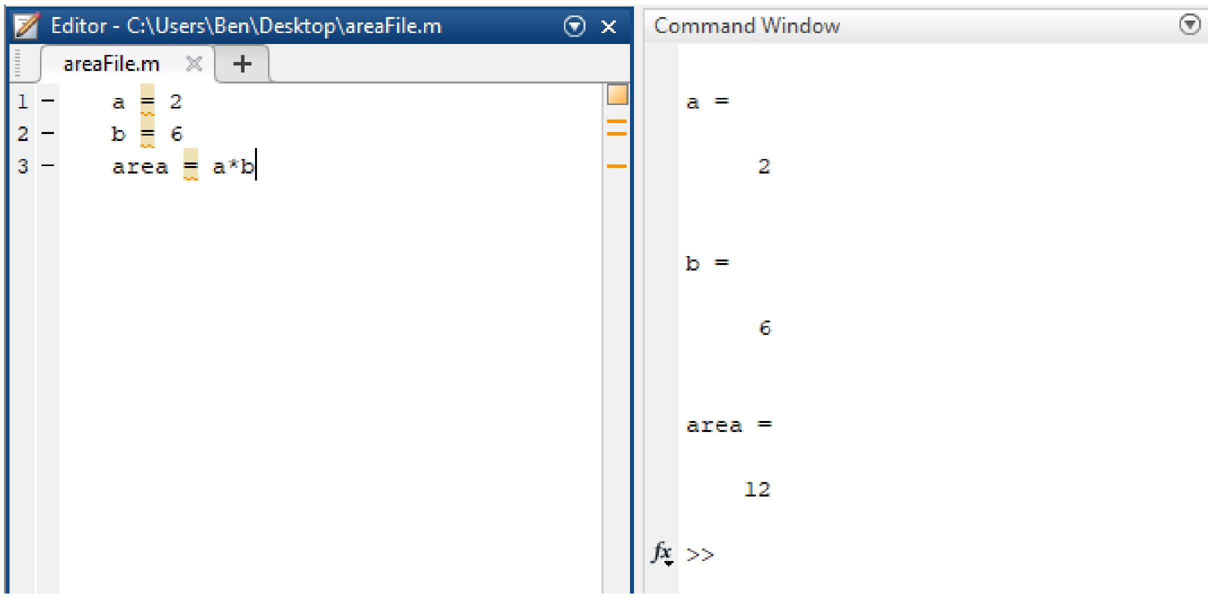


Figure 3: Run the program by click the highlighted button under the “Editor” ribbon tab (top).



```
a = 2
b = 6
area = a*b
```

```
a =

     2


b =

     6


area =

    12

>>
```

Figure 4: Using the m-file to solve for the area of a rectangle and displaying the results in the Command Window.

Go ahead and make a few changes to the m-file and run it again. Notice that the original information is still displayed in the Command Window. If we are going to run this program many times with different inputs, we would want to overwrite the old information and only display the up-to-date information in the Command Window. We can do this by using the clc and clear commands.

What are the clc and clear commands?

The `clc` command stands for clear Command Window. This command is used and placed in the m-file (it may be used in the Command Window too), and all the information preceding it gets cleared in the Command Window once the m-file is run. The `clear` command clears the assignment of all variables from a previous session (more specifically from the MATLAB “workspace”). Together, both of these commands provide for a smooth operating m-file, and it is considered good programming practice to make these commands the first two lines of your m-file. Example 1 shows the `clc` and `clear` commands being used in an m-file, and you will see this usage through the rest of the textbook.

How can I place comments in my m-file?

When you are writing a program, especially one that contains a large amount of information, placing notes (called comments) to yourself or anyone else that reads the code is beneficial. The MATLAB command to create a comment is the percent sign (`%`) followed by whatever text is required for the comment. Placing comments is essential and should be used to provide descriptions for the variables, expressions, etc. in the m-file. Example 1 shows the use of comments in an m-file. Note that comments do not affect the code and are not displayed in the Command Window (comments are non-executable code). To display a note in the Command Window, another command is required (see Lesson 2.4).

Example 1

Put a “comment” in MATLAB.

Solution

MATLAB Code example1.m

```
clc
clear

%This is a comment.
%In MATLAB comments show up as green and are not considered and executed.

%The following is a demonstration of the difference between a comment and
%    code.
a = 5  %Comments can also be at the end of a line of code, but not in the
       %    middle!
%b = 1
```

```
Command Window Output                                          Example 1
a =

     5

>>
```

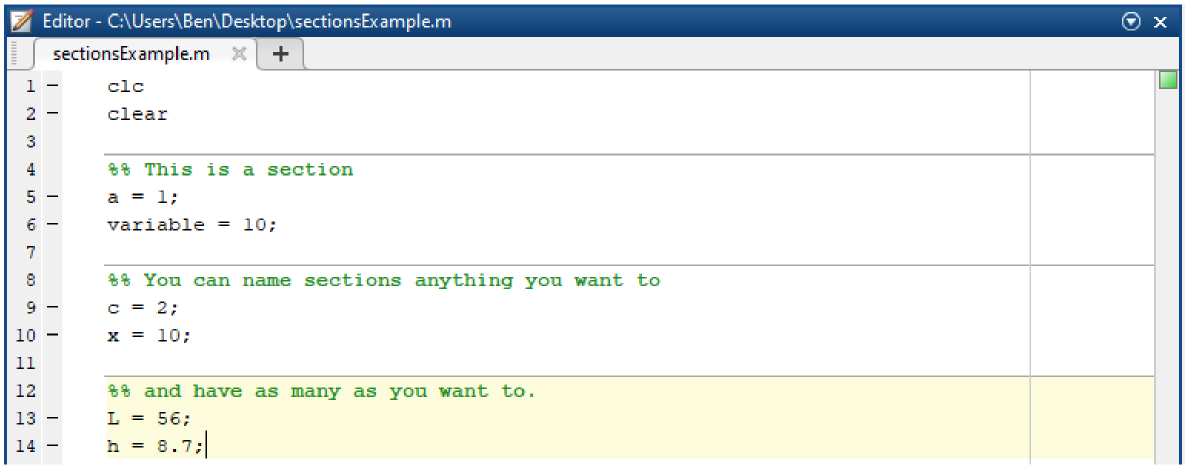
Can I separate my code into parts within the m-file?

MATLAB offers a special kind of comment called a section. You can think of sections like a paragraph in an essay. It is useful to visually separate paragraphs (sections), but they are part of the whole essay (program). To create a section in an m-file, simply type '%%'. Notice the mandatory space after the two comment characters. Sections allow you to do several useful things within your m-file, including:

1. using the "Run Section" feature, which only runs the current section > (wherever you last clicked). You can see this option right next to > the "Run" button in the Editor tab (Figure 5). Note that clicking > "Run" still runs the whole m-file as it normally would.

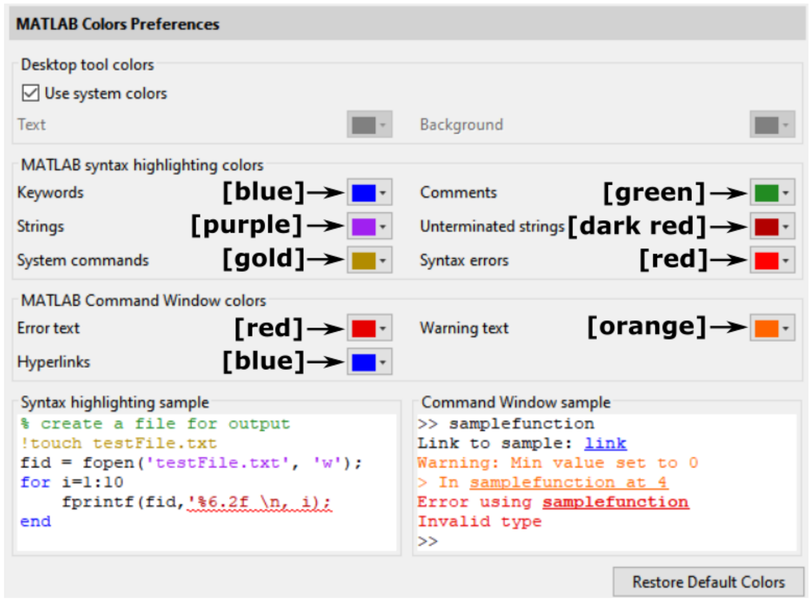
2. creating a more organized "published" m-file. You can see more > details about the published files and how to make them in Lesson > 1.7.

3. organizing your code within your m-file. Sections provide visual > divisions within the m-file as well. As seen in Figure 5, a > horizontal line gets placed at the top of a section, and the > active section is highlighted with a yellowish background.



```
clc
clear

%% This is a section
a = 1;
variable = 10;

%% You can name sections anything you want to
c = 2;
x = 10;

%% and have as many as you want to.
L = 56;
h = 8.7;
```

Figure 5: Using sections in an m-file.



Desktop tool colors

☑ Use system colors

Text | Background

MATLAB syntax highlighting colors

Keywords	[blue]→ 🟦	Comments	[green]→ 🟩
Strings	[purple]→ 🟪	Unterminated strings	[dark red]→ 🟥
System commands	[gold]→ 🟨	Syntax errors	[red]→ 🟥

MATLAB Command Window colors

Error text	[red]→ 🟥	Warning text	[orange]→ 🟧
Hyperlinks	[blue]→ 🟦		

Syntax highlighting sample

```
% create a file for output
!touch testFile.txt
fid = fopen('testFile.txt', 'w');
for i=1:10
    fprintf(fid,'%6.2f \n, i);
end
```

Command Window sample

```
>> samplefunction
Link to sample: link
Warning: Min value set to 0
> In samplefunction at 4
Error using samplefunction
Invalid type
>>
```

Figure 6: Default color highlighting preferences in MATLAB.

What does the color highlighting in the m-file mean?

By now, you have noticed that MATLAB highlights some text in different colors. In Figure 6, you can see the default color preferences in MATLAB which describe how MATLAB color codes different text. You can see comments are selected as green.

It is OK if you do not know what all of the terms mean (like "strings") in Figure 6 as we will cover these in later lessons and all examples are color-coded appropriately. This is just to introduce you to the idea of color-coding in MATLAB. You can find more information on the color settings page of MATLAB.

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Clear the command window	`clc`	`clc`
Clear one or all variables	`clear`	`clear`
Add an m-file comment	`%`	`% Any text can go here`

Multiple Choice Quiz

(1). The `clc` command is used to

(a) clear the command window.

(b) erase everything in the m-file.

(c) delete all saved (workspace) variables.

(d) save the existing m-file.

(2). The command to place a comment into the m-file is

(a) `%`

(b) `!`

(c) `;`

(d) `&`

(3). Which m-file is named correctly

(a) 1assignment.m

(b) assignment 1.m

(c) assignment1.m

(d) assignment/1.m

(4). The `clear all` command is used to

(a) clear the command window.

(b) erase everything in the m-file.

(c) delete all saved (workspace) variables.

(d) save the existing m-file.

(5). Clicking "Run" in MATLAB in an m-file with five sections runs

(a) the whole m-file.

(b) only the first section of the m-file.

(c) nothing because "Run" does not work with an m-file containing sections.

(d) only the last section of the m-file.

Problem Set

(1). The weight of an object can be found using

$$w = m \times g$$

where,

$w =$ weight (N)

$m =$ mass of the object (kg)

$g =$ gravity $\left(\text{m/sec}^2\right)$

Write an m-file to find the weight of a 30 kg Martian on earth (g=9.8 m/sec^2) and on Mars (g=3.7 m/sec^2). Make sure to suppress intermediate outputs and write comments in the m-file.

(2). You are asked to find the volume of a cylindrical storage tank. You know that the interior diameter is 3 feet and the length is 5 feet. Write a MATLAB m-file that finds the volume of this tank. Your boss needs the Command Window to look nice, so be sure to suppress intermediate outputs and only show the final answer in the Command Window.

(3). Congratulations! You've just been hired at The Pressure is on Us, Inc. Your first assignment is to write a MATLAB program that finds the pressure inside a cylindrical storage tank. You know that the tank has a 2.15 m interior diameter and is 4.55 m in length. You also know that the tank will hold 215 kg of an ideal gas at 400 K. The pressure in the tank can be found using:

$$p = \frac{mRT}{V}$$

where,

p is pressure (Pa)

m is mass (kg)

V is volume (m^3)

T is temperature (K)

R is the ideal gas constant, 287.05 $\dfrac{\mathrm{N \cdot m}}{\mathrm{kg \cdot K}}$

Follow the expected rules of display of Command Window and comments in m-file.

(4). A 12-volt battery and a switch are placed in parallel with the primary windings of a transformer. The secondary windings are placed in parallel with a resistor (See Figure A). Find the current traveling through the resistor after the switch is closed.

You may use

$\dfrac{n_s}{n_p} = \dfrac{V_s}{V_p}$, and $I = \dfrac{V}{R}$

where,

n_s = number of secondary windings

n_p = number of primary windings

V_p = primary voltage (V)

V_s = secondary voltage (V)

I = current (A)

R = resistance (Ω)

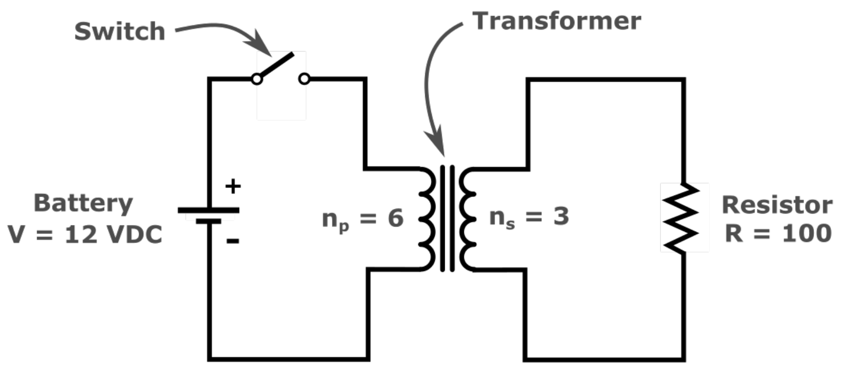


Figure A: Electrical Schematic for Exercise 4.

(5). The applied normal stress on an object is, in general, defined as the force acting on the object divided by the area the force is acting on. Assuming that a force, $F = 252.2$ lbs is acting in the direction as shown (see Figure B), find the normal stress in the body that has the top cross-sectional area of $a = 2$ in^2.

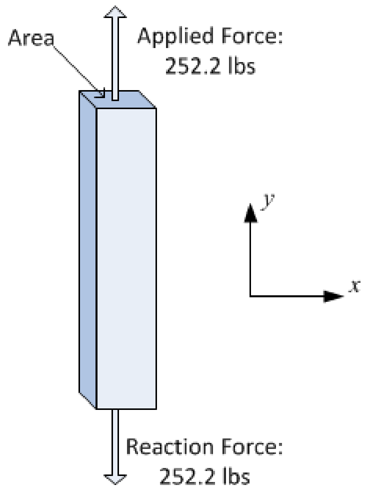


Figure 1: figure22e_stress

Figure B: Object with a force acting on its top face.

(6). When a normal axial load is applied to a thin plate with a hole, the nominal stress at the hole is amplified by a factor called the stress concentration factor, k_t. The maximum stress $\sigma_{\max}$is given by

$$\sigma_{max} = k_t \sigma_{nom}$$

where,

σ_{max} = the maximum stress in the body,

k_t = the stress concentration factor

σ_{nom} = the nominal stress.

For a thin plate with a center hole, the stress concentration factor k_t is

$$k_t \approx 3.0039 - 3.753\frac{D}{W} + 7.9735\left(\frac{D}{W}\right)^2 - 9.2659\left(\frac{D}{W}\right)^3 + 1.8145\left(\frac{D}{W}\right)^4 + 2.9684\left(\frac{D}{W}\right)^5$$

where,

W is the width of the plate

D is the diameter of the hole

If a force, F of 1230 lbs is applied to a thin plate (see Figure C), find the peak stress in the plate. The width of the plate is 30 inches, the diameter of the hole is 1.25 inches, and the plate thickness is 1.25 inches.

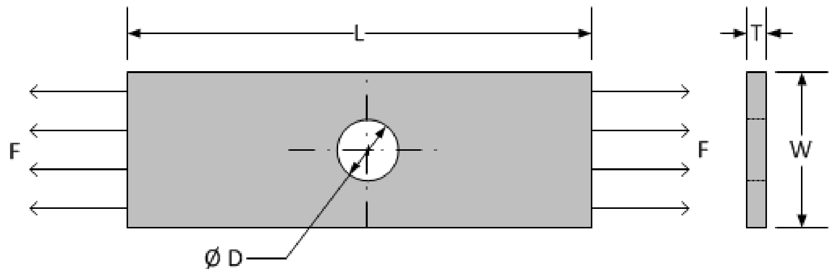


Figure 2: figure23e_wholestress

Figure C: Thin plate shown with applied load, F for Exercise 6.

Hint: To find the nominal stress you may use,

$A_{nom} = (W - D) \times T$

$\sigma_{nom} = \dfrac{F}{A_{nom}}$

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.6 – The Command Window

Learning Objectives

After reading this lesson, you should be able to:

1) *use the Command Window and understand its function,*
2) *control what appears in the Command Window,*
3) *access MATLAB documentation through the Command Window.*

What is the Command Window and how can I use it?

The Command Window (Figure 1) is the main window used to display the program output in MATLAB. We can also program directly into the Command Window; however, our code will not be saved in an m-file.

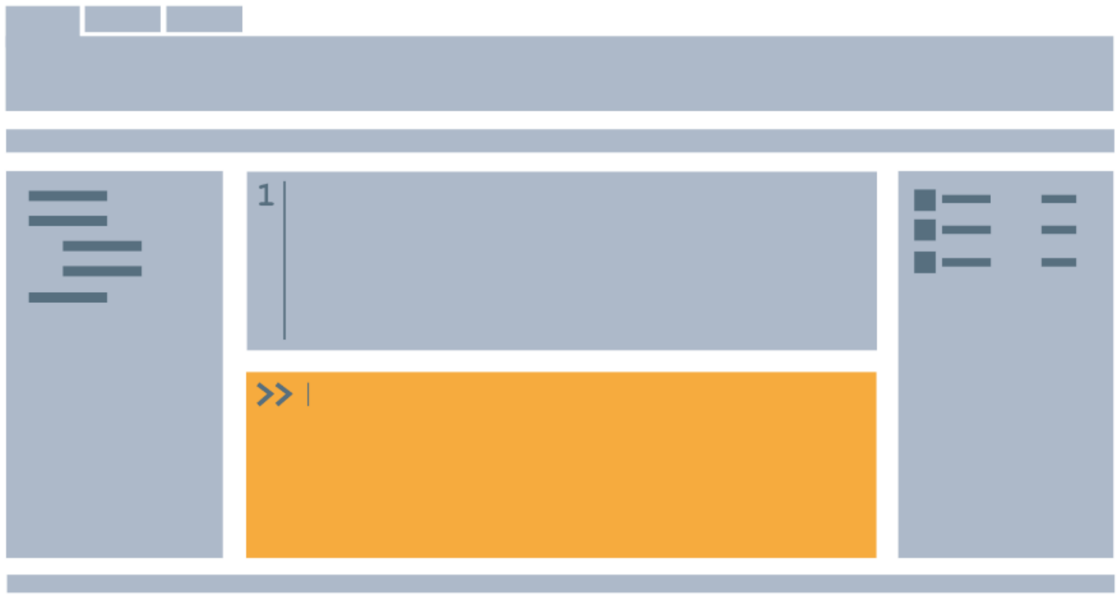
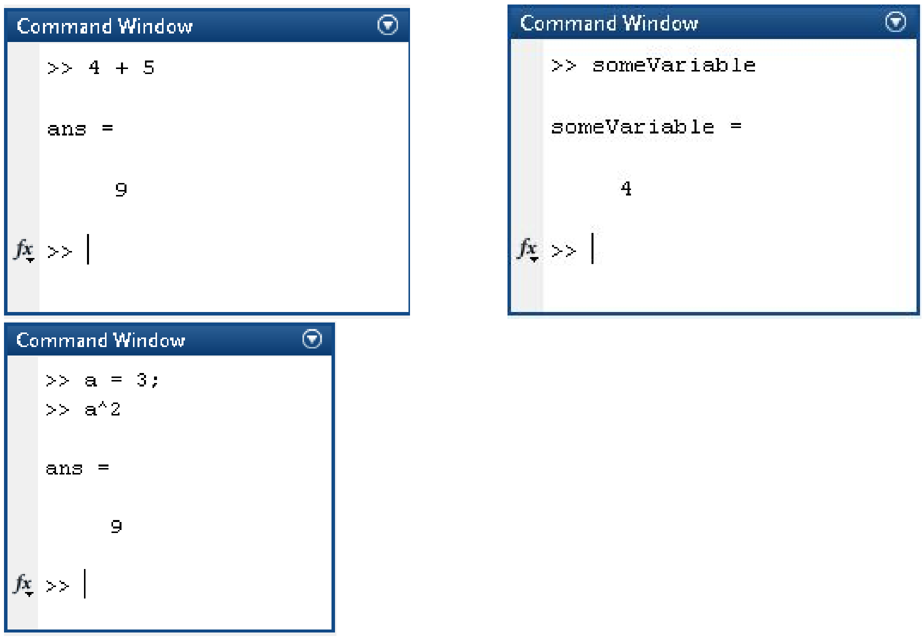


Figure 1: The default MATLAB window configuration is shown with m-file open. The Command Window is at the bottom center.

Table 1: Commonly used mathematical operators.

Operation	**Syntax**
Add	`+`
Subtract	`-`
Multiply	`*`
Divide	`/`
Power	`^`
Square Root	`sqrt()`

This means you can quickly test a few lines of code, use it as a calculator, or check the value of a variable (see Figure 2). This also means it can be a useful tool when debugging (find errors in the code) in your program. To use basic mathematical operators and numerals, just input the statement and hit the enter key. Table 1 shows a few common mathematical operators commonly used in MATLAB. Note that when the answer is displayed, it is assigned the name, `ans`, by default.



```
>> 4 + 5

ans =

     9

>> 
```

```
>> someVariable

someVariable =

     4

>> 
```

```
>> a = 3;
>> a^2

ans =

     9

>> 
```

Figure 2: Command Window used for a quick calculation (left), outputting the value of a previously defined variable (middle), and a quick and small program (right).

It is important to note that MATLAB does not understand what an equation is as known to you in a traditional sense. In MATLAB there is a variable name side and an expression side to any statement, and these sides are divided by an “equal to” sign. The variable name is always on the left side and the expression is always on the right side of an equal to sign. How this works is that MATLAB reads a variable name and attaches that variable name to what is on the right side of the equal to sign. This variable name is now associated with the number or expression on the right side of the equal to sign. If you do not assign the expression to a variable name, then MATLAB automatically assigns a default name, `ans`, to the expression.

How can I suppress outputs in the Command Window?

The Command Window will display the outputs of all executable code that is in the m-file. In most cases, the programmer may only want the final solution to be displayed, suppressing outputs of all other lines. Suppressing variables can also decrease the run time of your program (if it is long). To make this possible, MATLAB has a special character that can be added to any line of the m-file – the suppression character. This suppression is made possible by inserting a semicolon (;) at the end of any line of code. It is important to note that although the output of a line followed by the semicolon (;) is not displayed in the Command Window, the calculation in the line is still being done “behind the scene” by MATLAB. An example of using this character to suppress outputs from the Command Window is shown in Example 1.

✓ ***Important Note:*** Suppressing a line of code will only change whether it outputs to the Command Window or not: it does not stop MATLAB from performing the operation.

Example 1

Suppress variables from being output to the Command Window.

Solution

MATLAB Code example1.m

```
clc
clear

a = 2         %Creating an unsuppressed variable
b = 3;        %Creating a suppressed variable (note the semicolon)
area = a*b;   %There is no Command Window output from this line because
              %     we suppressed it.
area          %Demonstrating that “area” has been set to a*b
              %    (suppressing only stops the display output!).

%Note, if you are confused by which line is displaying “area = 6” in the
%    Command Window Output below, you should rerun this code yourself
%    and unsuppress all outputs. It should be immediately clear what
%    is happening.
```

Command Window Output Example 1

```
a =

     2


area =

     6

>>
```

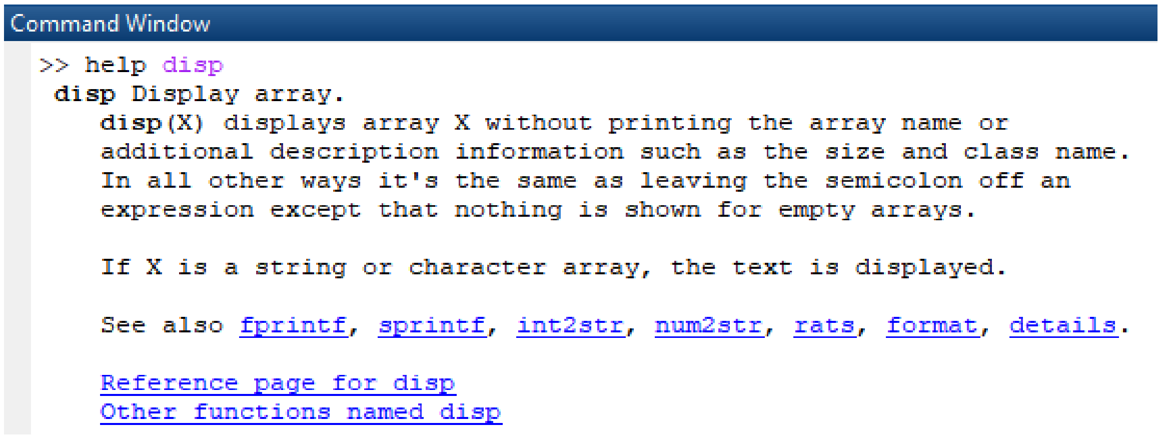
Can I view help from the Command Window?

If you cannot remember syntax/usage while you are programming, the `help` and `doc` commands can be used directly in the Command Window to pull up documentation on a specific MATLAB function or command.

The difference between `help` and `doc` is that `help` contains a summary of the documentation and displays directly in the Command Window (see Figure 3) while `doc` opens a new window with the full documentation page from MathWorks. Both are quick and easy ways to review documentation.

Can the Command Window do it all?

The Command Window does have limitations. In fact, it is rarely used to perform mathematical operations or for doing MATLAB programming. The main reasons for this are that it is difficult to change expressions without overwriting them and displaying useful information in the Command Window is cumbersome. For the purposes of this book, the main role of the Command Window is to only display the output information from a MATLAB file (also called an m-file). Although it is important to understand the inner workings of the Command Window, the m-file is the basis of MATLAB programming as described in Lesson 1.5.



```
>> help disp
 disp Display array.
    disp(X) displays array X without printing the array name or
    additional description information such as the size and class name.
    In all other ways it's the same as leaving the semicolon off an
    expression except that nothing is shown for empty arrays.

    If X is a string or character array, the text is displayed.

    See also fprintf, sprintf, int2str, num2str, rats, format, details.

    Reference page for disp
    Other functions named disp
```

Figure 3: Using the `help` command to retrieve documentation from the Command Window.

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Suppress an output (Command still processed)	`;`	`code;`
Get summary of documentation for a MATLAB command or function	`help`	`>>help disp`
Open documentation page for a MATLAB command or function	`doc`	`>>doc disp`
Add two variables	`+`	`a+b`
Subtract two variables	`-`	`a-b`
Multiply two variables	`*`	`a*b`
Divide two variables	`/`	`a/b`
Raise a variable to a power	`^`	`a^2`
Take the square root of a variable	`sqrt()`	`sqrt(a)`

Multiple Choice Quiz

(1). The MATLAB syntax to suppress output from the Command Window is

(a) `%`

(b) `&`

(c) `supp(' ')`

(d) `;`

(2). Using the Command Window, `2^(4^2)` will give the following output

(a) `16`

(b) `256`

(c) `512`

(d) `65536`

(3). Using the Command Window, `2^4^2` will give the following output

(a) `16`

(b) `256`

(c) `512`

(d) `65536`

(4). In the Command Window, if we enter:

```
 >>d = 5;
 >>a = d^2
```

then the output of the last line is

(a) `5`

(b) `10`

(c) `25`

(d) `Undefined function or variable 'd'.`

(5). In the Command Window, if we enter:

```
 >>a = 5;
 >>a = 6;
 >>a
```

then the output of the last line will be:

(a) `5`

(b) `5.5`

(c) `6`

(d) `11`

Problem Set

Use the Command Window to complete the following exercises.

(1). Find the output of b given that $a = 6$ and $b = 12a$.

(2). Using Ohm's law,

$$V = i \times R$$

find the electrical current i passing through a resistor of resistance $R = 2\times10^3\ \Omega$, and a voltage potential $V = 12$ V (DC).

(3). Find the area (in^2) of a right-angled triangle that has a base measurement of 4 inches and an adjacent angle of 32°.

(4). Find the lift force in Newtons of an airfoil at a constant velocity (V) of 35 m/s, and in a fluid environment with a density (ρ) of 1.247 kg/m^3. The airfoil has an exposed area (A_{exp}) of 2451 cm^2 and the coefficient of lift is found to be 0.81. For your solution, you may use the formula for the lift force as:

$$F_{\text{Lift}} = \frac{1}{2}\rho A_{\text{exp}} C_{\text{lift}} V^2$$

(5). Redo exercise 4 as follows. Find the lift force in Newtons using the same fluid density, exposed area, and lift coefficient as stated in Exercise 4, but choose the velocity first to be 25 m/s and then to be 50 m/s. Note that the values of fluid density, lift coefficient, and exposed area are already stored in MATLAB.

Module 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.7 – Publishing an m-file

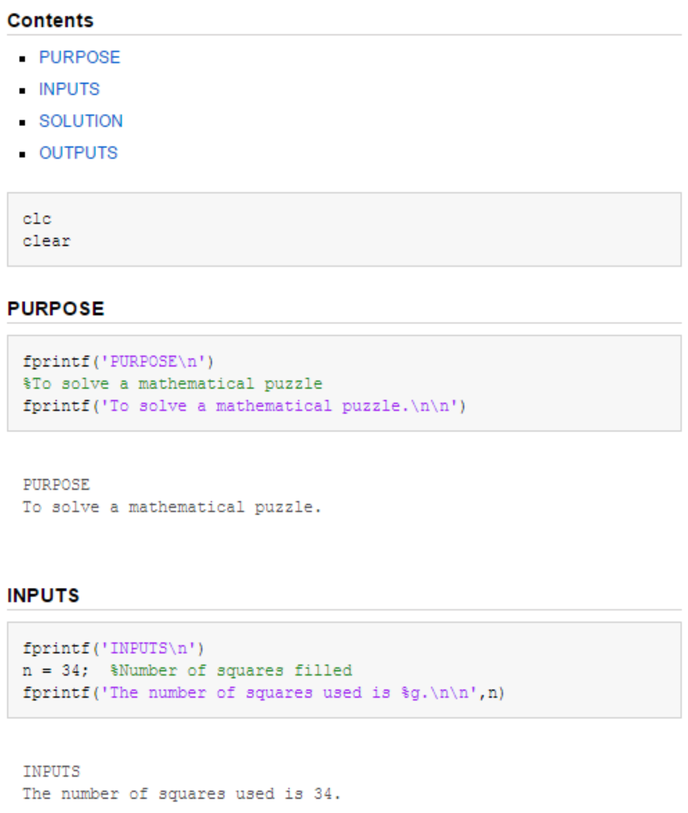
Learning Objectives

After reading this lesson, you should be able to:

1) *know why MATLAB is useful for engineers and scientists,*
2) *know how to get MATLAB on your computer,*
3) *comprehend the free alternatives to MATLAB.*

What does publishing do?

Publishing does several helpful things for you if you make good use of sections. Each of the following reflects the use of sections to split up different portions of the m-file (e.g., different exercises).



Contents

- PURPOSE
- INPUTS
- SOLUTION
- OUTPUTS

```
clc
clear
```

PURPOSE

```
fprintf('PURPOSE\n')
%To solve a mathematical puzzle
fprintf('To solve a mathematical puzzle.\n\n')
```

```
PURPOSE
To solve a mathematical puzzle.
```

INPUTS

```
fprintf('INPUTS\n')
n = 34;  %Number of squares filled
fprintf('The number of squares used is %g.\n\n',n)
```

```
INPUTS
The number of squares used is 34.
```

Figure 1: Example of published format.

A published file:

- Automatically links the sections contained in the m-file at the top of the published file. Click the link, and it takes you to that section.
- Can be saved in multiple file formats including .html and .pdf.
- Highlights each section and displays outputs directly below it.
- Shows comments as a description for each section (see below).

How can I publish in MATLAB?

In the toolbar, select the "PUBLISH" tab. Then click the "Publish" button on the far right of this figure.

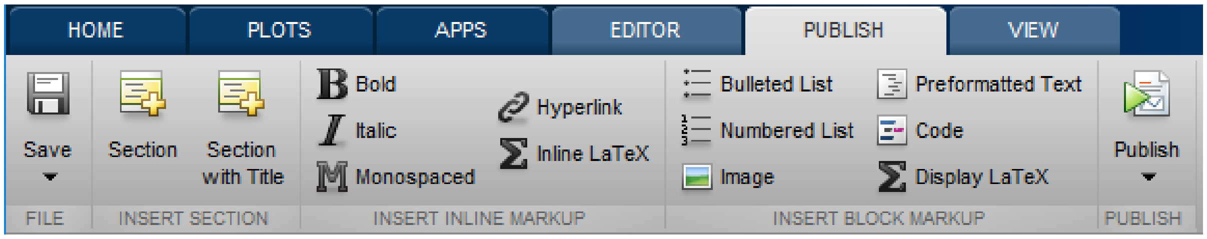


Figure 2: Publish tab in MATLAB toolbar.

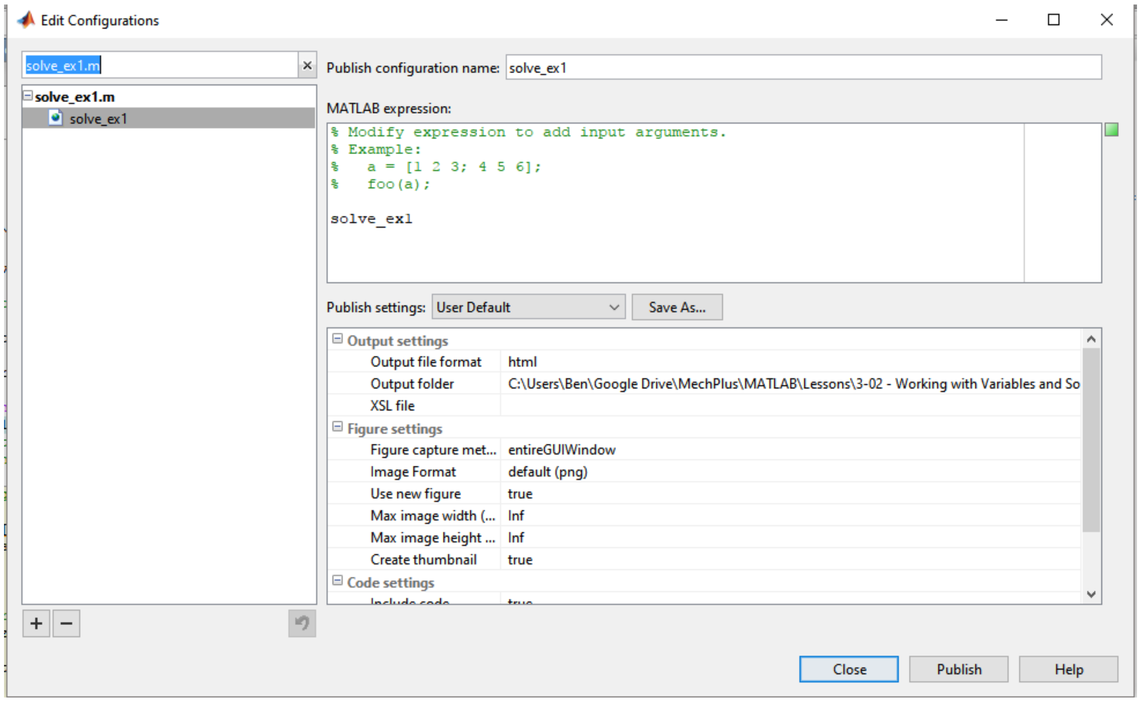
How can I get a PDF file of my published code?

You have two options to get a pdf. You can either change the publish settings to publish natively in pdf, or you can convert the html file into pdf using your internet browser.

1. *Publish as HTML (default) Then Convert to PDF* - Here are a couple of tips for converting from html to pdf. Keep in mind you can open the .html file in your web browser. So, you should start by doing that.

   For Chrome and Microsoft Edge, go to print, click “Change” to change printer, then click “Save as PDF” (for Chrome) or “Microsoft Print to PDF” (for Edge), and select where to save the .pdf file. Other browsers should have a similar procedure to save a .html file.

2. *Publish Directly as a PDF* - Click the arrow under Publish (seen in Figure 2) to open “Edit Configurations” (see Figure 3). Then change “Output file format” from html to pdf. Remember, it will be html by default.



Publish configuration name: solve_ex1

MATLAB expression:

```
% Modify expression to add input arguments.
% Example:
%   a = [1 2 3; 4 5 6];
%   foo(a);

solve_ex1
```

Publish settings: User Default

Output settings	
Output file format	html
Output folder	C:\Users\Ben\Google Drive\MechPlus\MATLAB\Lessons\3-02 - Working with Variables and So
XSL file	
Figure settings	
Figure capture met...	entireGUIWindow
Image Format	default (png)
Use new figure	true
Max image width (...	Inf
Max image height ...	Inf
Create thumbnail	true
Code settings	

Figure 3: Edit publishing configurations window where you can change the output file format.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Create a new section	`%%`	`%% section name`

Multiple Choice Quiz

The content of this page is intentionally blank

Problem Set

The content of this page is intentionally blank

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.1 – Variables and Naming Rules

Learning Objectives

After reading this lesson, you should be able to:

1) *define a mathematical variable,*
2) *define a programming variable,*
3) *determine legal and illegal variable names,*
4) *know the benefits of good practices for variable naming,*
5) *have guidelines for naming variables.*

What is a mathematical variable?

A mathematical variable is a number that we do not know yet and may have to solve for. For example, in the simple algebraic equation $2 + x = 3$, the mathematical variable is x. A mathematical variable can also essentially be a placeholder for substituting a variety/range of numbers. For example, we can substitute any range of numbers into x in the function $f(x) = x + 5$ to find the value of a function, $f(x)$. In a general sense, a variable varies its value. The value of a variable may be arbitrary, not specified, or even unknown.

What is a programming variable?

In computer programming, such as MATLAB, a programming variable connects the name of a variable and a specific storage location in the computer memory. For example, the variable x references/points to its allocated storage, which contains some information about the variable; e.g., the value you assigned to it. Figure 1 shows a simple graphical illustration of this concept.

Although a variable in computer programming can be used as a mathematical variable, it can also be used for many more applications such as substitution, information storage, iteration, value comparison, and much more. Do not worry if you are not sure how to do any of these in MATLAB yet; subsequent lessons will explain how to use variables for all of these.

How can I give my results a variable name of their own?

You have seen variables used in previous lessons, but you did not know the specifics about the concept or how to name them in MATLAB. To make sense of the information that you are receiving and inputting into MATLAB, you can assign names to the input, intermediate, and output variables. MATLAB allows you to name variables simply by typing the desired name followed by an equal to sign and then the operation. For instance, if you are using MATLAB to find the area of a square, you may type, `areaSq`, then press the “equal to” key followed by the operation of length times width. This will return the value of the area. You can also recall or use this expression in a later operation simply by typing in the variable name wherever it is needed. This is shown in Example 1.

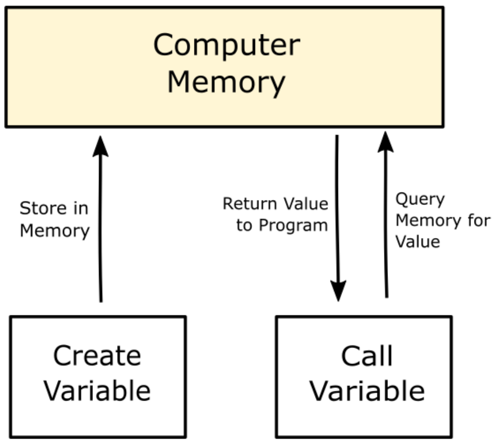


Figure 1: A simple visual representation of how variables are saved and called.

✓ ***Important Note:*** When naming variables, you cannot start the variable name with a number or use a space in the name. For example, `1cat` and `cat 1` are illegal variable names. Also, `cos` is an illegal variable name because it is used as a MATLAB function to calculate the cosine of an angle.

We will learn the MATLAB functions and commands later (throughout the rest of the book). MATLAB is case sensitive, and hence some programmers only use lower case script for variable names.

Example 1

Show examples of storing values in variables in MATLAB.

Solution

MATLAB Code example1.m

```
clc
clear

areaSq = 3*3

cubeSA = areaSq*6
```

```
Command Window Output                                    Example 1
areaSq =
     9

cubeSA =
    54

>>
```

Notice that in Example 1, the surface area of a cube, `cubeSA`, was found by using the predefined name `areaSq`, instead of physically typing the required dimensions to find the surface area. This is an example of recalling a previously named expression to make the current calculation easier and more readable.

What are some possible problems with naming an expression?

You have to be cautious when naming your expressions. Follow these rules for naming.

1. Do not begin a variable name with a number.
2. Do not put a space anywhere in the variable name.
3. Do not name a variable as a predefined MATLAB command or function name.

For example,

1. `1cat` is an illegal variable name, as it starts with a number.
2. `cat 1` is an illegal variable name as it has a space between characters.
3. `cos` is an illegal variable name as it is a predefined MATLAB function that calculates the cosine of an angle.

MATLAB reads inputs from the top to the bottom and from the left to the right of the page, similar to the way you might read a book. If you are using the same variable name multiple times, MATLAB will always use the last assigned value or expression to that variable name in its calculations. Example 2 below shows an example of replacing expressions.

Example 2

Examples of replacing an expression using the same name.

Solution

Note in the solution given below that although both `SA` and `a` are assigned values twice, the numeric values associated with those names are different. The second value of a replaces the first value and MATLAB uses this new value to calculate the next expression for `SA`. Both variables have been reassigned new numeric values or expressions.

MATLAB Code example2.m

```
clc
clear

a = 12*346
SA = a*6

a = 23*2
SA = 276
```

Command Window Output Example 2

```
a =
        4152


SA =
       24912


a =
    46


SA =
   276

>>
```

Are there benefits to good practices for variable naming?

Given that you stay within naming rules, you are free to use any variable name you wish. However, just because you can choose any variable name does not necessarily mean you should. Good variable names are essential to writing efficient and understandable code. Choosing your variable names wisely can have the following benefits:

1. *Readability and clarity* - It is easier to follow and understand programming code when proper variable naming techniques are followed.
2. *Debugging* - For more lengthy programming scripts, debugging (or troubleshooting) of code becomes more efficient and manageable.
3. *Collaboration* - If you are working with someone on a piece of code, or if someone needs to read and understand your code, it is important to name your variables in a clear way. That someone could also be you trying to figure out or reuse your code five years from now.

Are there some guidelines for variable naming that I can follow?

This section contains a more explicit set of guidelines for naming your variables in MATLAB. These are strongly recommended; however, MATLAB will not give any errors if these are not followed. Failing to follow good naming conventions, though, can make looking at a simple program seem intimidating and frustrating. Also, note that different computer programming languages may have different naming conventions.

1. A variable covering a large scope (used across a wide range of the program) should have more specific and meaningful names.
    - **Good:** `voltageDrop`, `pullForce`, `outputTemperature`
    - **Bad:** `vd`, `forP`, `To`
2. A variable covering a small scope (used across a short range of the program) should have short, disposable names. This guideline generally applies to loop counters or dummy variables.
    - **Good:** `i`, `j`, `elem`
    - **Bad:** `looponeiteration`, `temporaryVariable10`
3. Use CamelCase with leading lower case letters. Note the use of underscores between words (e.g., box_height = 5) is also common; however, CamelCase will be used throughout this textbook.
    - **Good:** `pressureSensorOutput`, `boxHeight`, `width`
    - **Bad:** `pressuresensoroutput`, `Boxheight`, `WiDtH`

4. Avoid negating boolean (value of true or false) variable names (no double negatives). The concept of boolean variables will be covered in Module 5: Conditional Statements.

    - **Good:** `isGood`, `isMax`, `error`
    - **Bad:** `isNotGood`, `isNotMax`, `noError`

5. Do not make the variable name very long. Also, the maximum length of variable names is limited to 63 characters. You will have to use your own judgement beyond this constraint. In general, it must be long enough to be descriptive, yet short enough to be memorable and useful.

    - **Good:** `avgPartStress`, `isTankLightOn`
    - **Bad:** `averageStressInPartThatIsConnectedToTheOtherPart`, `isTheFirstLightOnTopOfTheTankFlashingGreen`

The following example shows a short MATLAB script with *badly selected variable names*. For this example do not worry about the function of this specific MATLAB m-file script. We have included examples and explanations for each guideline to elucidate each point more clearly and hopefully impress them on your memory.

MATLAB Code badFormatting.m

```
clc
clear

% GUIDELINE 1
%Explanation: Looking at these calculations, it becomes hard to keep
%             track of what each variable means since the names are
%             not descriptive.
a = 2;
z = 5/a;
r = 8;
q = z*r;

% GUIDELINE 2
%Explanation: Although you do not know what for loops are (covered in
%             Module 8: Loops ), this is one of the most common
%             examples for a "short scope". The variable name
%             "iterationvar7" makes  the code messy due to its
%             unnecessary length.
for iterationvar7 = 1:5
      iterationResult = iterationvar7*a + iterationvar7;
end

% GUIDELINE 3
%Explanation: This makes the program harder to read because there is
%             no discernible marker between words.
pressuresensoroutput = 5.5;

% GUIDELINE 4
%Explanation: It is unclear exactly what is true. Is it good or not?
%             Is it maximum or not?
isNotGood = true;
isNotMax  = false;

% GUIDELINE 5
%Explanation: Having very long variable names also makes the code
%             difficult to read even if other guidelines, like
%             CamelCase, are followed. Additionally, variable names
%             that are this long are rarely necessary for description
%             and can usually be shortened.
voltageReadingFromSecondSensor = 10;
MAXimumnumberofreadings        = 20;  %This violates both guidelines
                                      %    3 and 5 and compounds the
                                      %    problem.
```

In the code below, we rewrite our first MATLAB script example to put our guidelines into practice. Again, for this example do not worry about the function or meaning of this specific MATLAB m-file program. Note that the added white space and alignment further enhance readability.

MATLAB Code goodFormatting.m

```matlab
clc
clear

%Note: We feel that the good variable names in this example should be
%      obviously better and do not require further explanations beyond
%      what we have already given.

% GUIDELINE 1: A variable covering a large scope (used across a wide range
%    of the program), should have more specific and meaningful names.
scale = 2;

% GUIDELINE 2: A variable covering a small scope (used across a short range
%    of the program) should have short, disposable names.
for reading = 1:5
      readingResult = reading*scale + reading;
end

% GUIDELINE 3: Use CamelCase with leading lower case letters.
pressureSensorOutput = 5.5;

% GUIDELINE 4: Avoid negating boolean (value of true or false) variable names
%    (no double negatives).
isSensorGood = true;
isVoltMax    = false;

% GUIDELINE 5: Do not make the variable name very long.
voltageReadingSensor2 = 10;
%Note, this could be further abbreviated to the following depending
%    on your preference
voltReadSensor2 = 10;
```

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage

Store a value in a programming variable	`validName`	`validName = 6`
		`validName2 = 'some text'`

Multiple Choice Quiz

(1). A correct name for a variable is

(a) `1arearec`

(b) `area rec`

(c) `area_rec`

(d) `cos`

(2). What is the mistake in the following code (m-file is saved with the name *exercise_2.m*)?

```
clc
clear

five = 5
first variable = 6
```

(a) The m-file name is invalid.

(b) One of the variables is called (referenced) before it is defined (assigned a value).

(c) One of the variable names is invalid.

(d) None of the above

(3). An incorrect name for a variable is

(a) `cat1`

(b) `cat_1`

(c) `cat_cos`

(d) `1cat`

(4). The following variable follows the rules of camelCase.

(a) `AreaSquare`

(b) `areaSquare`

(c) `areasquare`

(d) `Areasquare`

(5). The following program is given to you. What is the value of the variable c?

```
clc
clear

a = 5
b = 6
c = a*b
c = a*b^2
b = 7
```

(a) 30

(b) 35

(c) 180

(d) 245

Problem Set

(1). Enumerate the benefits of good naming practices of variables.

(2). Enumerate guidelines for variable naming. Give examples of each enumeration.

(3). Enumerate illegal variable names in MATLAB. Give an example of each.

(4). Write a program with proper names and good practices that calculates the inertial force in a mass with an acceleration. The value of the mass and acceleration are inputs (you choose these values) and the inertial force is the output.

(5). Write a program with proper names and good practices that calculates the current through a resistor. The value of the resistance and voltage are known inputs (you choose these values) and the current through the resistor is the output. Remember, $I = \dfrac{V}{R}$.

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.2 – Characters and Strings

Learning Objectives

1) *know the programming concept of a character/string,*
2) *use strings in MATLAB.*

What is a character?

A character, or "char" in MATLAB, is a data type (we will cover what "data types" are in Lesson 2.5) that is as simple as a one single unit: a text character. This can be a symbol, letter, punctuation, space, etc.

In MATLAB, you can turn text into characters by using single quotes (e.g., `'s'`, `'g'`). The text will then turn purple signifying that MATLAB now recognizes the text as characters, and has stored it that way.

✓ ***Important Note:*** If it is a character, MATLAB does not recognize it as numbers, variables, functions, etc. For example, `'2+2'` as a string of characters does *not* equal `4`. It is only text and will not be calculated.

Example 1

Assign the characters “1”, “n”, and “x” to three different variables.

Solution

```
MATLAB Code                                          example1.m

clc
clear

%Defining some very simple examples of characters
exChar1 = '1'
exChar2 = 'n'
exChar3 = 'x'
```

Without recognizing the text as a character, MATLAB will think it is a variable and ask the computer for its value. You can verify this for yourself by removing the quotes on `exChar2` or `exChar3` (e.g., `exChar2 = n`).

```
Command Window Output                                 Example 1

exChar1 =
    '1'


exChar2 =
    'n'


exChar3 =
    'x'

>>
```

What is a string?

A string is an array of characters (alphanumeric) and can be a word, a sentence, or any other combination of characters. Strings are often used to display and describe outputs in the Command Window. In MATLAB, a string is established by the characters that are set within two single quotes (example: `'This is an example of string theory'`). Strings can be assigned to a standard variable just like any other expression as shown in Example 2. They also have array “properties” like referencing specific portions of the string as seen in Example 2. We will cover this in more detail in Lesson 2.6: Vectors and Matrices. For now, just know that this is possible with strings.

Example 2

Assign the text "Take 5!" to a variable.

Solution

```
MATLAB Code                                                    example2.m
clc
clear

myString = 'Take 5!';  %Assigning a piece of text to a variable

%Note: these reference commands will be covered shortly in the lesson on
Vectors and Matrices
myString(1)     %The first letter in the character array (index 1)
myString(2:4)   %Range of letters in the character array (index 2 through 4)
```

```
Command Window Output                                           Example 2
ans =
    'T'


ans =
    'ake'

>>
```

What makes characters/strings special?

Example 3 demonstrates some of the things we have talked about so far in this lesson. When using the characters, MATLAB will not recognize the text as variables, numbers, etc. This is important to remember going forward. The use of a variable in a string must be called in a special way, which we will learn in Lessons 2.3 and 2.5. Writing a variable name inside a string, as shown in the last line of Example 3, will not work!

Example 3

Show examples of the properties of strings.

Solution

MATLAB Code example3.m

```
clc
clear

%Example of a string (collection of text)
exString1 = 'keep on going on'

%Numbers as Characters
exString2 = '2+2'  %Defining the numbers as characters

%NOTE:   Try defining either string without the quote marks and see
%        what MATLAB does! For exString1, it will give an error
%        because there is no variable with that name. For exString2,
%        the answer will be 4.

%Variables as Characters
a = 1             %Defining an arbitrary variable
b = a             %Note: b = a = 1
exString3 = 'b' %This is just the text "b": not the variable b = 1!
```

Command Window Output Example 3

```
exString1 =
    'keep on going on'

exString2 =
    '2+2'

a =
     1

b =
     1

exString3 =
    'b'

>>
```

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Use text in the program	`'some text'`	`exStr = 'some text'`
Reference part of a string	`exStr(1:4)`	`ans = 'some'`

Multiple Choice Quiz

(1). Strings are designated by using

(a) `/ /`

(b) `' '`

(c) `\ \`

(d) `" "`

(2). What is the Command Window output of the following code?

```
clc
clear
base = 2;
T = '3*4'
```

(a) `base = 2;`

(b) `T = '3*4'`

(c) `T = 12`

(d) Nothing, there will be an error.

(3). Which of the following *will* perform the addition of the numbers 3 and 4 in MATLAB?

(a) `sum = (3 + 4)`

(b) `sum = '3 + 4'`

(c) `sum = '3' + '4'`

(d) All of the above

(4). What is the Command Window output of the following code?

```
clc
clear
height = 2;
H = 'height'
```

(a) `H = 'height'`

(b) `H = 2`

(c) `Undefined function or variable 'height'.`

(d) None of the above

(5). To store the phrase “Land of the Free” in a variable, the correct code is

(a) `myVar! = 'Land of the Free!'`

(b) `myVar = Land of the Free`

(c) `myVar = 'Land of the Free'`

(d) `'Land of the Free'`

Problem Set

1. Store the phrase “Sweet Home Alabama” in a variable and suppress the variable. Then call that variable and observe its output.

2. Write mathematical operations `7*(5+4)` as a string and store in the variable `myString`. Next, complete the operations as mathematical numbers (answer should be 63) and store in the variable `hwAnswer`.

3. Use the characters `%` and `'` in a string. Hint: since these are special characters in strings in MATLAB, use doubles of each character (e.g., `%%` and `''`).

4. Get the first name from the string `'John Smith'`, and store it in the variable `firstName`.

5. Get the last name from the string `'Jane Doe'`, and store it in the variable `lastName`.

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.3 – Working with Strings

Learning Objectives

1) *concatenate (join) strings,*
2) *analyze and manipulate strings.*

In this lesson, we will discuss a few common and useful string manipulation MATLAB functions and techniques. However, there are many more MATLAB string manipulation functions available than we can cover in this lesson.

How do I join two strings together?

The programmer can take multiple strings (that are assigned to variables) and combine them – this is called *concatenation.* An example of concatenation would be to combine several words to form a single sentence. Example 1 shows the use of string concatenation.

Example 1

Assign the following two strings to two variables named, `str1` and `str2`, in a new m-file.

string 1 = `'Strings can'`

string 2 = `' tie a knot.'`

Concatenate the two input strings to form a single sentence and show this sentence in the Command Window.

```
MATLAB Code                                                    example1.m
clc
clear

str1 = 'Strings can'     %String one
str2 = ' tie a knot.'    %String two: notice the space at the beginning of
                         %    this string

joinedStr = [str1 str2]  %Concatenating string 1 and string 2
```

```
Command Window Output                                          Example 1
str1 =
    'Strings can'

str2 =
    ' tie a knot.'

joinedStr =
    'Strings can tie a knot.'

>>
```

Notice the brackets used in the concatenation line (last line) of the m-file. This is required to join the two strings together in this method (there are other methods we do not cover here). The same process is also used if more than two strings are to be joined together.

It is important that you do not try to concatenate strings and standard numerals or numeric variables as it will not properly work. You can only join strings together to form new strings.

How do I search and count strings?

This section gives you some examples of how MATLAB can analyze strings. We will focus on only two functions, but you can refer to MATLAB documentation for characters and strings to see a full list.

If you are presented with a lengthy piece of text and would like to see if a word or phrase is contained within that lengthy piece of text, you can use the `contains()` function. Another example of analyzing a string is to count how many times a word or phrase occurs within the text by using the `count()` function.

Important Note: The `contains()` function will return a Boolean (true or false) value. It will return a 1 for true if it did find a match, and a 0 for false if it did not find a match.

Example 2

You are given the sentence, *That gray dog is chasing that yellow dog.*

Check whether the sentence (string) contains the word (string) *gray* and count how many times the word (string) *dog* occurs in the sentence.

MATLAB Code example2.m

```
clc
clear

someText = 'That gray dog is chasing that yellow dog.';  %Assigning text to
                                                          %    a variable
isDogGray    = contains(someText,'gray')    %Checking if the text contains
                                            %    the word 'gray'
dogsInvolved = count(someText,'dog')        %Counting the number of times
                                            %    the word 'dog' appears
```

Command Window Output Example 2

```
isDogGray =
  logical
   1

dogsInvolved =
     2

>>
```

How do I make a whole string lower or upper case?

This section and the following one are a few examples of manipulating strings with MATLAB. You can see a full list of the different functions for string manipulation MATLAB offers by reviewing the MATLAB documentation for characters and strings.

Sometimes it is necessary to convert a string to all lower case or all UPPER case letters. This may be particularly useful when comparing two user input strings to each other such as `'Yes'` and `'yes'`. MATLAB offers the `lower()` and `upper()` functions to accomplish this.

Example 3

You are given the sentence, *How Is It Going, SmOOth cAt?*

Convert the sentence (string) to all upper case letters (characters) and store it in a variable. Repeat the process for lower case.

MATLAB Code example3.m

```
clc
clear
%Assigning a piece of text to a variable
someText    = 'How Is It Going, SmOOth cAt?'
someTextLow = lower(someText)  %Converting text to all lower case
someTextUp  = upper(someText)  %Converting text to all upper case
```

Command Window Output Example 3

```
someText =
    'How Is It Going, SmOOth cAt?'

someTextLow =
    'how is it going, smooth cat?'

someTextUp =
    'HOW IS IT GOING, SMOOTH CAT?'

>>
```

Can I split a string into its component pieces?

Often, it is useful to be able to split (or parse) strings when processing data and this can be done by using the MATLAB function `strsplit()`. Data has to be separated (delimited) by characters like a comma, space, or tab for `strsplit()` to work.

Example 4

Split a single string, *'sup,5,3,yes,no,54.0'*, into multiple strings (pieces) based on the comma delimiter. Next, split the string, *'MATLAB does some cool stuff!'*, into multiple strings (words) based on the space delimiter.

Solution

```
MATLAB Code                                                    example4.m

clc
clear

dataInput1 = 'sup,5,3,yes,no,54.0'     %Assigning text to a variable
dataSplit1 = strsplit(dataInput1,',')  %Separating the string by commas

dataInput2 = 'MATLAB does some cool stuff!'  %Assigning text to a variable
dataSplit2 = strsplit(dataInput2, ' ')       %Separating the string by
                                             %    spaces ( )
```

```
Command Window Output                                          Example 4

dataInput1 =
    'sup,5,3,yes,no,54.0'

dataSplit1 =
  1×6 cell array
    {'sup'}    {'5'}    {'3'}    {'yes'}    {'no'}    {'54.0'}
```

```
Command Window Output (continued)                              Example 4

dataInput2 =
    'MATLAB does some cool stuff!'

dataSplit2 =
  1×5 cell array
    {'MATLAB'}    {'does'}    {'some'}    {'cool'}    {'stuff!'}

>>
```

✓ ***Important Note:*** The data that is read from each cell array is a string even if it is shown as a number (e.g., `'1'`). (Note: Instead of using `strsplit()`, you may also use the function `split()`, which outputs directly to a string array instead of a cell array. However, `split()` is only available in MATLAB version R2016b and newer.)

Example 5

Split a single string into multiple strings based on a delimiter.

Solution

The delimiter in the input string below is ; (semi-colon).

```
MATLAB Code                                                  example5.m
clc
clear

dataInput = '1;2;3;4;5;6'              %Assigning text to a variable

dataSplit = strsplit(dataInput,';')   %Separating the string with semi-colons

firstElem = dataSplit{1}               %Outputting first item from the split
```

```
Command Window Output                                        Example 5
dataInput =

    '1;2;3;4;5;6'

dataSplit =

  1×6 cell array

    {'1'}    {'2'}    {'3'}    {'4'}    {'5'}    {'6'}
```

```
Command Window Output (continued)                            Example 5
firstElem =

    '1'

>>
```

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Join (concatenate) strings together	`[]`	`['string1', 'string2']`
Count the number of occurrences of a pattern inside another string	`count()`	`count('the dog is in the yard', 'the')`
Check whether or not a string contains the pattern string	`contains()`	`contains('the dog is running', 'dog')`
Convert all letters in a string to upper case	`upper()`	`upper('tHis TeXt')`
Convert all letters in a string to lower case	`lower()`	`lower('tHis TeXt')`
Split a string into pieces output as a cell array	`strsplit()`	`strsplit('these;are;words', ';')`

Task	Syntax	Example Usage
Split a string into pieces output as a string array	`split()`	`split('these are words')`

Multiple Choice Quiz

(1). The output of `cat = ['cat' 'dog']` is

(a) `catdog`

(b) `cat dog`

(c) `cat_dog`

(d) `CatDog`

(2). The output of the last line is

```
clc
clear
aa  = 2;
as  = num2str(aa);
cat = ['cat'   as]
```

(a) `cat2`

(b) `cat 2`

(c) `Undefined function or variable 'as'.`

(d) `cat as`

(3). To count the number of instances that one string occurs in another string (pattern), one should use the function

(a) `contains()`

(b) `count()`

(c) `check()`

(d) `howmany()`

(4). To separate the phrases or pieces of a string, one should use the function

(a) `phrases()`

(b) `pieces()`

(c) `comp()`

(d) `strsplit()`

(5). To search a string to see whether in contains another string (pattern), one should use the function

(a) `search()`

(b) `contains()`

(c) `ismember()`

(d) `find()`

Problem Set

(1). Concatenate the following strings:

```
cm1 = 'My name is'
cm2 = 'Slim Shady.'
```

How would you change the string(s) so that there is proper spacing between each word?

(2). Count how many times the character *'e'* occurs in the string `'Programming is important for mechanical engineers.'`

(3). Check whether each of the following strings contains the corresponding patterns.

String: `'My name is Slim Shady.'`	Pattern: `'name'`
String: `'My name is Slim Shady.'`	Pattern: `'Eminem'`
String: `'The United States of America'`	Pattern: `'Unit'`

(4). Store the sentence "Watch out!" in a variable, and then convert it to all upper case letters.

(5). Split the string `'The,United,States,of,America'` into its pieces using the `strsplit()` function and the comma delimiter.

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.4 – Inputs and Outputs

Learning Objectives

1) *get/store user defined inputs with the `input()` function,*
2) *output messages to the Command Window with `disp()` and `fprintf()` functions.*

How can I get input from the user?

When a computer programmer hands over a completed program to whoever is going to use it (called the end-user), they typically do not ask the end-user to modify the code. Rather they might prompt the user for the required inputs and then let the program run. The input() function prompts the user in the Command Window to enter a value for a designated variable. The value of this variable will then be used in the program.

Function: Prompts the user for an input.
`input()`

Function Inputs:
Variable value which is requested, `x`
String to prompt the user for the variable input, `'the value of x is'`

Example Usage:
`x = input('the value of x is')`

Until the user enters a value for a designated variable in the Command Window, that variable is not defined, and the program will not continue to run. (Note, this is why m-files cannot be published with `input()` function.) These undefined variables and the pause in the program can cause some troubleshooting issues. If you find that your program is not running properly, check the lower-left corner or the Command Window for the message: `'waiting for input'` and assign values as needed. To cancel the program from running altogether, hit `Ctrl+c` while in the Command Window (while it is the "active" window). The use of the `input()` function is being shown in Example 2.

How do I display notes in the Command Window?

Like comments in the m-file, displaying information about a program in the Command Window is also important. Information such as the programmer's name, the purpose of the program and the variables involved in the calculation may be needed to make the output information clear. MATLAB uses several functions to display information in the Command Window. This section will cover the two basic functions that can be used in the m-file to display information in the Command Window. These functions are the disp() and fprintf() functions.

Function: Displays a single string or variable.
`disp()`

Function Inputs:
Test String, `'Hello'`, OR, Expression name, `x`

Example Usage:
```
disp('Hello')
disp(x)
```

Example 1

Output the string "*The height of the beam is*" and the variable height = 12.63 using two separate disp() functions. Then concatenate the given string and variable into a single string and output it using disp().

Solution

MATLAB Code example1.m

```
clc
clear

height = 12.63;  %Defining the variable

%Displaying the string and variable separately
disp('The height of the beam is')  %Outputting a string input with disp()
disp(height)                       %Outputting a variable input with disp()

%Concatenating both the string message and the variable, height.
message = ['The height of the beam is ', num2str(height)];
%Then displaying the concatenated string.
disp(message)
```

Command Window Output Example 1

```
The height of the beam is
   12.6300

The height of the beam is 12.63
>>
```

The **`disp()`** function displays either a string *or* a variable in the Command Window. **`disp()`** does not require or allow you to manually create new lines. It will do this automatically (see the Command Window output in Example 1). In Example 1, we should several use cases for **`disp()`**. Notice that you can output a variable with a message, but it requires you to concatenate them into one string.

Function: Can display from more than one input (both variables and a description).
`fprintf()`

Function Inputs:
String, `'Note about the number'`
Expression name, `x` (can be more than one)
Conversion character, (`%g`, `%f`, `%d`, etc.)

Example Usage: `fprintf('\n note about the number %g \n', x)`

The `fprintf()` function is used to make the Command Window look more organized and easier to follow. The text (string) that is inside the quotes will be displayed to provide a description of a variable, and the `%g` is where the variable will be placed in the string. The `%g`, known as a conversion character, sets the display format for the number (e.g., exponential or decimal format) and tells `fprintf()` what type of data to expect there (e.g., number or string). The `fprintf()` function does not start a new line to display its information, so we use the `\n` line character. This line character starts a new line, and the `\n` is not displayed in the Command Window. The last item one needs to provide for the `fprintf()` function is the name of the variable to be displayed. Example 2 shows the `fprintf()`functions being used in an m-file. See Table 1 for commonly used formatting characters that are used inside `fprintf()`.

Table 1: Formats to be used with the `fprintf()` function.

Task	Formatting Character	Example of Placement
Add a line space (enter key)	`\n`	`fprintf('\n note %g',a)`
Default display	`%g`	`fprintf('\n note %g',a)`
Fixed point display	`%f`	`fprintf('\n note %f',a)`
Single character display	`%c`	`fprintf('\n note %c',a)`
Scientific notation display	`%e`	`fprintf('\n note %e',a)`
String display	`%s`	`fprintf('\n note %s',a)`
Adjust field width (MATLAB controls this automatically)	Any whole number (Using 6 as an example)	`fprintf('\n note %6f',a)`
Adjust precision (change amount of numbers after the decimal point)	Any whole number (Using 2 as an example)	`fprintf('\n note %.2f',a)`

Understanding the correct method for naming and placing variables in the m-file as well as the order in which MATLAB reads these inputs are the core foundations for writing any program in MATLAB. It is very important to make the program as easy for the user or reader as possible. Using the display `disp()` and `fprintf()` functions in conjunction with the suppression character (`;`) and `clc` command is vital. These functions provide a good foundation for any m-file.

Example 2 shows the `input()`, `disp()`, and `fprintf()` functions being used together in a single program. The usage of each character is shown with a generic `fprintf()` function call. Table 1 shows other formatting characters that are commonly used with the `fprintf()` function.

Example 2

Calculate the age of the user based on the inputs of what year they were born and the current year. Output the inputs (birth and current years) and the result (age) in the Command Window with a description of the numbers.

Solution

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%This program shows an example of the input function.
fprintf('This program shows examples of the input and output\n')
fprintf('functions. We calculate the age of the user based on the\n')
fprintf('inputs of what year they were born and the current year.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Prompt user for year of birth and current year
birthYear = input('In what year where you born?   ');
year = input('What is the current year?   ');

%----------------------------- SOLUTION ----------------------------
%Calculating the age of the user (this year)
age = year - birthYear;

%----------------------------- OUTPUTS -----------------------------
fprintf('\nOUTPUTS\n')
%Displaying the input variables
fprintf('You were born in %g. The current year is %g.\n',birthYear,year)
%Displaying the calculation and age of the user
fprintf('Therefore, you are/will be %g years old this year.\n\n',age)
```

The Command Window output for Example 2 is shown below, which after the user entered the desired input.

Command Window Output Example 2

```
PURPOSE
This program shows examples of the input and output
functions. We calculate the age of the user based on the
inputs of what year they were born and the current year.

INPUTS
In what year where you born?   1993
What is the current year?   2019

OUTPUTS
You were born in 1993. The current year is 2019.
Therefore, you are/will be 26 years old this year.

>>
```

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Ask the user for an input	`input()`	`input('What is your age?')`
Display a note in the Command Window	`disp()`	`disp('note')`
Display a variable with a note in the Command Window	`fprintf()`	`fprintf('note %g',a)`

Multiple Choice Quiz

(1). To display, Question 2, in the Command Window, the correct syntax is

(a) `disp(Question 2)`

(b) `display('Question 2')`

(c) `disp('Question 2')`

(d) `Question 2`

(2). Although not typically necessary, one can use which of the following to add a new blank line?

(a) `disp(' ')`

(b) `disp('\n')`

(c) `disp(\n)`

(d) `disp()`

(3). What is the output of the following code? Assume the user inputs the number 10 when prompted.

```
clc
clear

force = input('What is the force on the beam in kN?');
area = 5;
P = force/area;
fprintf('pressure = %.0f', P)
```

(a) `pressure = 2`

(b) `pressure = 2.0`

(c) `2`

(d) `0.5`

(4). The correct `fprintf()` usage to display the variable `myString = 'strings are fun'` is

(a) `fprintf('I think %s, but they do not agree!\n')`

(b) `fprintf('I think %.3f, but they do not!\n')`

(c) `fprintf('I think %e, but they do not!\n',myString)`

(d) `fprintf('I think %s, but they do not!\n',myString)`

(5). To output the variable `length = 8.40235` with two decimal places, use

(a) `fprintf('The length is %g\n',length)`

(b) `fprintf('The length is %2d\n',length)`

(c) `fprintf('The length is %2f\n',length)`

(d) `fprintf('The length is %.2f\n',length)`

Problem Set

(1). Print out the following string in the Command Window using `fprintf()`.

`cm1 = 'My goal is to become an engineer.'`

(2). Find the product of any two numbers by using `input()` to prompt the user to enter the numbers in the Command Window. Use `fprintf()` to display the product of the numbers in the Command Window.

(3). Find and output in the Command Window the kinetic energy of an object of mass, m traveling at a specified instantaneous velocity, v. Use the `input()` function to prompt the user to enter the values of the mass (kg) and velocity (m/s).

Use `fprintf()` to display the numeric value of the kinetic energy (J) of the object. Be sure to add a good description for the `input()` and `fprintf()` functions to avoid any miscommunications (that is, display the units as well). Hint: The formula for kinetic energy is

$$\text{KE} = \frac{1}{2}mv^2$$

.

(4). Write a program in MATLAB that outputs the maximum tensile normal force that a steel cylindrical tube can handle without breaking. A factor of safety n and inner and outer diameters (meters) of the tube are specified by the user. The `fprintf()` function must be used where needed for a detailed description. The other specifications and formulas are given below.

$$\sigma_{\text{steel}} = 245 \times 10^6 \text{Pa},$$
$$\sigma_{\text{Safe}} = \frac{\sigma_{\text{steel}}}{n},$$
$$F_{\text{Safe}} = \text{Area}\left(\sigma_{\text{Safe}}\right),$$
$$\text{Area} = \frac{1}{4}\pi\left(\left(D_{\text{outer}}\right)^2 - \left(D_{\text{inner}}\right)^2\right).$$

where,

D_{inner} is the inner diameter (m) of the tube,

D_{outer} is the outer diameter (m) of the tube,

σ_{steel} is the yield strength of steel (does not require the `input()` function).

(5). The monthly payment on a car loan is given by the formula

$$PMT = \frac{LA * IPM}{1-(1+IPM)^{-NM}}$$

where,

PMT = monthly payment in dollars,

LA = loan amount in dollars,

IPM = interest rate given as a fraction per month (Note the units),

NM = number of monthly payments (Note the units).

Write a MATLAB program that calculates the monthly payment for buying a car, based on the loan amount (dollars), length of the loan (years) and interest rate (annual percentage rate). Three inputs are assigned at the beginning of the worksheet through a variable assignment:

1. Loan amount entered in dollars, `LA`
2. Length of loan entered in integer years, `NY`, and
3. Interest rate entered in annual percentage rate (APR), `APR`.

The output is:

1. The monthly payment on the car.

Display and print with an explanation at least the following in the Command Window.

- A short description of the problem,
- Loan amount in dollars,
- Length of loan in integer years,
- Interest rate in annual percentage rate,
- Monthly payment in dollars.

Example to compare our MATLAB program against:

For a loan of \$14,800 at 6.75% annual percentage rate (APR) for a 4 year term,

$LA = \$14,800$

$IPM = APR/(12*100) = 6.75/(12*100) = 0.005625$

$NM = NY*12 = 4*12 = 48$ months

Hence,

$PMT = \dfrac{14800*0.005625}{1-(1+0.005625)^{-48}}$

$= \$352.69$

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.5 – Data Types

Learning Objectives

1) *account for the programming concept of a data type,*
2) *convert between data types,*
3) *check the data type of any variable in MATLAB.*

What is a data type?

In computer programming, a data type is classification (or type) of the data you use to program. That is, MATLAB categorizes each variable or any piece of data into different groups. For example, MATLAB will treat a numeric integer (1, 5, 99, etc.) differently than an alphabetical character (“a”, “U”, etc.). Recall from Lesson 2.4 (Inputs and Outputs), we had to include in the `fprintf()` function what data type we were giving it (e.g., `%s` for string).

MATLAB refers to the different data types as fundamental MATLAB classes: MATLAB uses “data type” and “class” interchangeably. For someone familiar with object-oriented programming (C, C#, Java, etc.), this may be a little confusing. However, for the scope of this lesson, do not overthink this concept.

What are the MATLAB data types?

While the range of data types that a programming language supports varies with the language, MATLAB solely works with six fundamental data types. Fundamental MATLAB data types are:

- Numeric - 1, 2, -54, 4.56
- Logical - True, False
- Character - ‘g’, ‘M’, ‘.’, ’ ’
- Cell - Can contain any data type
- Table - Data stored in tabular format
- Struct - Data related to groups using data containers called fields

While there are multiple data types available to MATLAB users, this course will focus on the most common: numeric, logical, and character/strings.

Why are data types important?

The types of data (and therefore the “data types”) you will use in MATLAB will vary depending on their function and application. Some MATLAB functions only accept a specific data type, while other functions accept multiple forms of data. For example, MATLAB function `fprintf()` requires a character data type input (e.g., `fprintf('hey there!')`), while the `mean()` function requires numeric data types (e.g., `mean(1.0, 2.5, 2.0, 3.0, 6.6)`).

How do I check the data type of a variable?

Let’s say you want to know the data type of a variable. For example, you want to probe the variable `myVariable` and find out its data type. This is achieved using the `class()` MATLAB function. Note that in the following example, the output is `double`, which is a sub-data type of the numeric class (data type).

Example 1

Store the number 5.678 in a variable, and check the data type (class) of the variable.

Solution

```
MATLAB Code                                                    example1.m
```

```matlab
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To check the data type (class) of a variable.
fprintf('To check the data type (class) of a variable.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
myVariable = 5.678;  %Defining a variable
fprintf('The value stored in the variable we want to check is %g.\n',...
          myVariable)

%------------------------------ SOLUTION -----------------------------
varClass = class(myVariable);   %Class function to find the data type of
                                %    the variable

%------------------------------ OUTPUTS ------------------------------
fprintf('\nOUTPUTS\n')
fprintf('The data type (class) of the variable is %s.\n\n',varClass)
```

As a side note, notice that there are three consecutive dots used in Example 1 so that the `fprintf()` function can be written on two lines. This will work in many other cases, which you will see in future examples. One important rule for using this is that it cannot be used in the middle of a string or a number.

```
Command Window Output                                           Example 1
```

```
PURPOSE
To check the data type (class) of a variable.

INPUTS
The value stored in the variable we want to check is 5.678.

OUTPUTS
The data type (class) of the variable is double.

>>
```

You can ask MATLAB if the particular variable is a specific data type, where MATLAB will respond with either yes, true (1), or no, false (0). This is achieved using a variety of MATLAB data type identification functions.

One of these functions, `ischar()`, is used if you would like to see if a particular variable is a character data type. In the following example, a numeric double input of `5.678` into the `ischar()` function returns `false`, which means that

5.678 is not a character. On the other hand, notice that `isnumeric()` will return a `true`.

Example 2

Conditionally check whether a variable is the char data type or a numeric data type.

Solution

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To conditionally check the data type of a variable.
fprintf('To conditionally check the data type of a variable.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
myVariable = 5.678;      %Variable definition
fprintf('The value stored in the variable is %g.\n',myVariable)

%----------------------------- SOLUTION ----------------------------
check1 = ischar(myVariable);     %Asks if variable is a "char" data type
check2 = isnumeric(myVariable); %Asks if variable is a "numeric" data type
```

MATLAB Code (continued) example2.m

```
%----------------------------- OUTPUTS -----------------------------
fprintf('\nOUTPUTS\n')
fprintf('The test for "char" is %s. ',string(check1))
fprintf('The test for "numeric" is %s.\n',string(check2))
```

The use of `string()` in the last two lines of the code is to convert the logical value (0 or 1) to "false" or "true", respectively, for readability. You can see the output of this below. We will cover logical data types in Lesson 5.1; so, you do not need to worry about this conversion right now.

Command Window Output Example 2

```
PURPOSE
To conditionally check the data type of a variable.

INPUTS
The value stored in the variable is 5.678.

OUTPUTS
The test for "char" is false. The test for "numeric" is true.
>>
```

Can I convert between data types?

MATLAB provides functions to convert between most data types. Many of these have straightforward names like `num2str()`, which converts a number to a string, or character array. Others are simply the name of the MATLAB class like `char()` or `double()`. We will cover more of these functions as appropriate in the following lessons. You can review MATLAB documentation for a full list of data type conversion functions.

Example 3

Convert the number 5.678 into a string.

Solution

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To convert a number to a string.
fprintf('To convert a number to a string.\n\n')
```

MATLAB Code (continued) example3.m

```
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
myVariable = 5.678;                %Defining a variable
varClass1 = class(myVariable);     %Displaying the data type of the variable
fprintf('The value stored in the variable is %g.\n',myVariable)
fprintf('The data type (class) of the variable is %s.\n\n',varClass1)

%------------------------------ SOLUTION ----------------------------
myStr = num2str(myVariable);  %Converting the variable from numeric to
                              %    string data type
varClass2 = class(myStr);     %Displaying the data type of the new
                              %    converted variable

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The variable after conversion is %s\n',myStr)
fprintf('The data type (class) of the variable is %s.\n',varClass2)
```

```
Command Window Output                                    Example 3
PURPOSE
To convert a number to a string.

INPUTS
The value stored in the variable is 5.678.
The data type (class) of the variable is double.

OUTPUTS
The variable after conversion is 5.678
The data type (class) of the variable is char.
>>
```

Another common example of where data types become important is that you cannot directly join a string and a number to form a new string. To concatenate a number with other strings, you must first use the `num2str()` function on the number. The `num2str()` function converts a number to a string, thus making the number capable of joining with other strings. Conversely, to convert a string to a number use `str2num()` function. Though it should be noted that this will only work if the characters are numbers. The difference between *numbers* and *characters that are numbers* was covered in more detail in Lesson 2.2.

Example 4

Input the following two strings and expression, `str1`, `str2`, and `n`, respectively into a new m-file.

```
str1 = 'Strings can tie'

str2 = 'or more knots'

n = 3
```

a) Convert the variable, `n` to a string and concatenate these three strings into a single sentence, “String can tie 3 or more knots.”, and output it to the Command Window using the `fprintf()` function.

b) Convert the variable, `n` back to a number and output the sentence “You are making at least 3 knots.” to the Command Window using the `fprintf()` function.

Solution

MATLAB Code example4.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To concatenate a number with strings.
fprintf('To concatenate a number with strings.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
str1 = 'Strings can tie ';  %String one
str2 = ' or more knots.' ;  %String two
n = 3;                      %Number

fprintf('String 1: %s\n',str1)
fprintf('String 2: %s\n',str2)
fprintf('The number: %g\n\n',n)

%----------------------------- SOLUTION ----------------------------
str3 = num2str(n);        %Converting the number to a string
str  = [str1 str3 str2];  %Concatenation of strings
num  = str2num(str3);     %Converting the string back to a number

%----------------------------- OUTPUTS -----------------------------
fprintf('\nOUTPUTS\n')
fprintf('a)  %s\n',str)   %Printing the concatenated string
fprintf('b)  You are making at least %g knots.\n',num)  %Printing to show
                                                        %  converted string
```

You can see that the m-file in Example 4 uses three pieces to make a full sentence, where one of the pieces is defined as a number, n, and requires the use of the num2str() function to be concatenated with the two other strings. It must be converted to a string because all three pieces (variables) must be strings, or more precisely, they must all be of the char data type. The variable str3, which stores the character 3, is then converted “back” to a number data type, as a demonstration. This is accomplished using the str2num() function. The variable str3 is then a number and can be used with the fprintf() function and the %g format.

Command Window Output Example 4

```
PURPOSE
To concatenate a number with strings.

INPUTS
String 1: Strings can tie
String 2:  or more knots.
The number: 3


OUTPUTS
a)  Strings can tie 3 or more knots.
b)  You are making at least 3 knots.
>>
```

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Check the data type (class) of any variable	`class()`	`class(a)`
Convert a numeric value to a double precision number	`double()`	`double(a)`
Check if a variable is a numeric data type or not	`isnumeric()`	`isnumeric(a)`
Convert a variable to a char data type (equivalent to a string)	`char()`	`char(a)`
Check if a variable is a char data type or not	`ischar()`	`ischar(a)`
Convert a number to a string	`num2str()`	`num2str(a)`
Convert a string to a number	`str2num()`	`str2num(a)`

Multiple Choice Quiz

(1). The `num2str()` function

(a) converts a number to string

(b) converts a string to a number

(c) concatenates numbers and strings

(d) concatenates strings

(2). The function to check the data type of a variable in MATLAB is

(a) `type()`

(b) `class()`

(c) `datatype()`

(d) `var()`

(3). To convert a scalar variable to a string (character data type), one should use the function

(a) `char()`

(b) `scal2str()`

(c) `var2str()`

(d) `str2num()`

(4). What data type do the conversion characters `%g`, `%f`, and `%e` correspond to in the `fprintf()` function?

(a) `char`

(b) `numeric` (includes `double`)

(c) `logical`

(d) `struct`

(5). To check if the stored value of a variable is numeric or not, one should use the function

(a) `ischar()`

(b) `isnumeric()`

(c) `isspace()`

(d) `isletter()`

Problem Set

(1). Define the variables in MATLAB given below and check the data type (class) of each one. Display the value of each variable along with its data type using `fprintf()`.

```
myString = 'Test string'

myNum = 100
```

(2). Using the variable `age = 30`, form a string that says, “My age is 30.” Store this string in a variable called `myAge`.

(3). Using the functions `ischar()` and `isnumeric()`, check the variables given below to see whether or not they are the character or numeric data type. You should do both checks for each variable.

```
units = 'Newtons'
```

```
myNum = 50
```

(4). Start by checking the data type of the variable `var = 83`. Next, convert `var` to the char data type, store this string in a variable called `strVar`, and check the data type of this variable. Display both variables and their corresponding data types using `fprintf()`.

(5). Using the variable `age = '30'`, convert it to a number and store it in a variable named `ageConv`. Use `fprintf()` to print both variables with proper formatting.

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.6 – Vectors and Matrices

Learning Objectives

1) *define vectors and matrices in MATLAB,*
2) *manipulate vectors and matrices effectively with common MATLAB syntax,*
3) *reference strings as vectors (arrays).*

Why is the program called MATLAB?

MATLAB stands for Matrix Laboratory, and as the name implies, MATLAB is designed to solve problems using matrices. When MATLAB was originally designed, the basis of the program was to solve all problems using matrix algebra. Even if the user is conducting simple scalar operations, MATLAB will store these scalar values in a matrix of the size 1×1. By storing all values as matrices and using solvers based on matrix algebra, MATLAB programmers were able to achieve high efficiency, and hence low computational solving time.

In previous lessons, all the programs for the examples were written using scalar operations and variables. (Scalar means "one number".) We now know that MATLAB has stored these scalar values as matrices for all required m-file operations. This lesson will discuss the use of matrices and matrix operations in

MATLAB. If you are not comfortable with basic linear algebra concepts, see the Primer on Linear Algebra in Appendix A at the end of this textbook.

What is a matrix?

A matrix is a special organized way of representing groups of numbers. This organized form is similar to a grid or table, but where each cell, or "element", has a standard reference. Matrices have two dimensions/directions called rows and columns. (Two dimensional matrices can be stacked to form three dimensional matrices, but we do not need to worry about this for now). In Figure 1, we see the standard notation for referencing an element in a matrix.

Figure 2 shows an example of how a matrix of numbers is often written on paper. In the bottom right of the figure we have noted the size of the matrix (the number of rows and columns). However, this is not mandatory or standard to write with each matrix explicitly.

✓***Important Note:*** When reporting the size of a matrix, the number of rows is followed by the number of columns (e.g., 2×3 in Figure 2).

Do I need to know any special types of matrices?

Yes, there are several uses for a special type of matrix in linear algebra. Examples include an identity matrix, a symmetric matrix, and an upper-triangular matrix. MATLAB even has dedicated functions to create some of these matrices. For example, `eye()` creates an identity matrix of any size you input. We will discuss these special matrices more in Module 8.

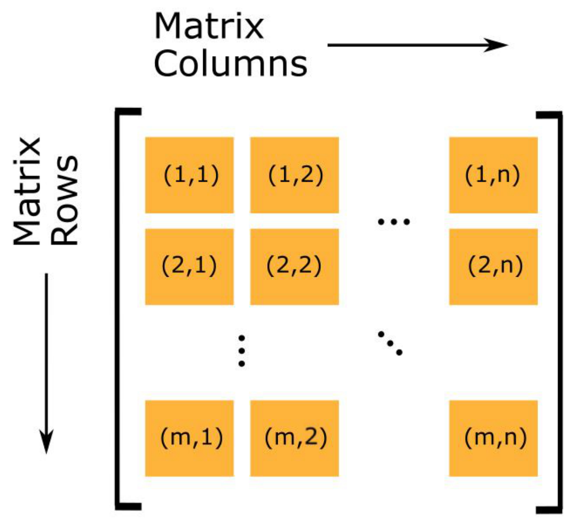


Figure 1: Generalized example of a matrix (size: m × n).

$$\begin{bmatrix} 12 & 0 & 43 \\ 5 & 33 & 1.2 \end{bmatrix}_{2\times 3}$$

Figure 2: A matrix with 2 rows and 3 columns (size: 2 × 3).

What is a vector?

A vector is a category of matrices, a one-dimensional matrix. This means it has only one row (a row vector - see Figure 3) *or* one column (a column vector - see Figure 4). As mentioned before, the numbers to the bottom right of the vector denote the size of the vector.

✓ ***Important Note:*** Since a vector is just a type of matrix, the rules, notation, and syntax for matrices in MATLAB also apply to vectors.

You can see an example of a vector containing numbers in Figure 5. Do not confuse programming (MATLAB) vectors with a general Euclidean vector, which has a magnitude and direction (e.g., 3 meters at 45 degrees). Also note that vectors and matrices are types of arrays. Arrays are the most general type of ordered data in programming.

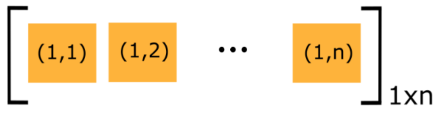


Figure 3: Generalized example of a row vector (size: $1 \times n$).

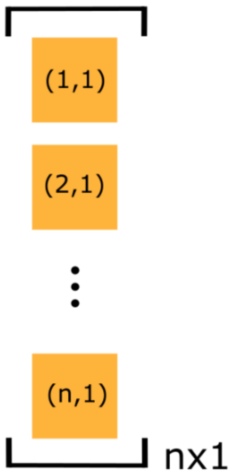


Figure4: Generalized example of a column vector (size: $n \times 1$).

$$[4 \quad 10 \quad 99]$$

Figure 5: A vector with 1 row and 3 columns (size: 3×1).

How do I define a vector or a matrix in MATLAB?

Matrices that contain all their elements in one row or all their elements in one column are called row and column vectors, respectively. If a matrix has more than one row and one column, it is a multidimensional matrix (the smallest size of a multidimensional matrix is 2×2).

When defining a matrix in MATLAB, we must differentiate between rows and columns. To separate individual row elements, at least one space is required. To separate each matrix row, a semicolon is required. The nomenclature is the same for a matrix of any size.

The steps to input a matrix of any size are:

1. Define the matrix using a legal named variable.
2. Define the matrix by using a set of brackets [].

3. Inside the brackets, matrix rows are input with each element separated by at least one space.

4. Use a semicolon (;) to denote that the row input is complete.

5. Now, repeat steps 3 and 4, until all the rows are entered.

An example of how to enter a matrix is given next in Example 1.

Example 1

Input the following two matrices ($[A]$ and $[B]$) into a MATLAB m-file.

$$[\mathrm{A}] = \begin{bmatrix} 1 & 2 & 6 \\ 2 & 65 & 1 \end{bmatrix} \text{ and } [\mathrm{B}] = \begin{bmatrix} 2 & 3 \\ 2 & 1 \\ 9 & 3 \end{bmatrix}$$

Solution

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To show examples of how to effectively reference matrices.
fprintf('To show examples of how to effectively reference matrices.\n\n')

%----------------------------- SOLUTION ----------------------------
A = [1 2 6;2 65 1];

B = [2 3;
     2 1;
     9 3];

%Note: Since MATLAB does not care about whitespace, we can format our
%    matrix B in a slightly more intuitive way.

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
disp('A =')
disp(A)
disp('B =')
disp(B)
```

```
Command Window Output                                          Example 1
PURPOSE
To show examples of how to effectively reference matrices.

OUTPUTS
A =
     1     2     6
     2    65     1

B =
     2     3
     2     1
     9     3

>>
```

You can see the steps for inputting a matrix being followed in Figure 1. Note the differences between inputting a scalar and a matrix.

What are some basic functions and commands for matrix manipulation?

MATLAB has implemented many built-in functions that make basic matrix manipulations simpler and these are shown in Table 1.

Table 1: Several commonly used matrix commands and operators. The input to the listed commands is the matrix $[A]$ and $[B]$ (where applicable).

Task	**Syntax**	**Explanation**
Isolating a matrix element	`A(r#,c#)`	Calls a matrix element by a row # and column #
Maximum dimension of a matrix/vector	`length(A)`	Outputs the largest dimension of a vector/matrix
Size of a matrix	`size(A)`	Has two outputs: the number of rows and the number of columns of a matrix.
Last element of a dimension	`A(end,1)`	References/indexes the last element in that dimension.
Matrix referencing shorthand	`A(:,1)`	Yields the entire first column. The ":" references "all of that dimension".
Matrix referencing shorthand	`A(1:20,1)`	Yields rows 1 through 20 in column 1 of matrix $[A]$.

Task	**Syntax**	**Explanation**
Develop a zero matrix	`zeros(m,n)`	Outputs a matrix of size m × n with all zeros elements
Create an identity matrix	`eye(n)`	Outputs an identity matrix of size n × n
Develop a ones matrix	`ones(m,n)`	Outputs a matrix of size m × n with all elements equal to one
Create an arbitrary square matrix	`magic(n)`	Outputs an arbitrary square matrix of size n × n

Example 2

Create an arbitrary matrix, and find its dimensions. Then create a vector from a subset of this matrix and calculate the number of elements in it.

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To show examples of how to effectively reference vectors and matrices.
fprintf('To show examples of how to effectively reference vectors\n')
fprintf('and matrices.\n\n')
```

MATLAB Code (continued) example2.m

```
%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
A = magic(4);  %Creating a special 4x4 (square) matrix
disp('A =')
disp(A)

%----------------------------- SOLUTION ----------------------------
matrixDimensions = size(A);       %This returns a row vector of the form
                                  %    [rows, columns]
myVector       = A(1,:);          %Creating a vector from the first row
                                  %    of the defined matrix
vectorLength = length(myVector);  %Finding the length of the vector

%Note: The function length can be useful for vectors since we do not need.
%      to know whether it is a row or a column vector.

%----------------------------- OUTPUTS -----------------------------
fprintf('\nOUTPUTS\n')
fprintf('The matrix A has %.0f rows and %.0f columns.\n\n',...
         matrixDimensions(1),matrixDimensions(2))

disp('myVector =')
disp(myVector)
fprintf('The length of myVector is %.0f.\n\n',vectorLength)
```

Command Window Output Example 2

```
PURPOSE
To show examples of how to effectively reference vectors
and matrices.

INPUTS
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

OUTPUTS
The matrix A has 4 rows and 4 columns.

myVector =
    16     2     3    13

The length of myVector is 4.

>>
```

Example 3

Create an arbitrary matrix $[A]$. Then create two vectors, `vec1` and `vec2`, from subsets of that matrix.

The vector, `vec1`, should contain the *first* three elements of the first row of the matrix $[A]$. The vector, `vec2`, should contain the *last* three elements of the first row of the matrix $[A]$.

Solution

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To create vectors from a subsets of an existing matrix.
fprintf('To create vectors from a subsets of an existing matrix.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
A = magic(4)  %Creating a 4x4 (square) matrix

%------------------------------ SOLUTION ----------------------------
vec1 = A(1,1:3);         %Creating a vector from the FIRST 3 elements of
                         %    the first row of matrix A
vec2 = A(1,end-2:end);   %Creating a vector from the LAST 3 elements of
                         %    the first row of matrix A

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
disp('vec1 =')
disp(vec1)
disp('vec2 =')
disp(vec2)
```

Command Window Output Example 3

```
PURPOSE
To create vectors from a subsets of an existing matrix.

INPUTS

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Command Window Output (continued) Example 3

```
OUTPUTS
vec1 =
    16     2     3

vec2 =
     2     3    13

>>
```

How can I reference strings as vectors?

In Lesson 2.2, we discussed characters and strings. In MATLAB and many other programming languages, strings (or arrays) of characters, can be referenced the same way as a vector. See Example 4 for usage. This can be especially useful if we only want to look at a particular part of a string/text.

Example 4

You are given the sentence, *This is an example string.* Assign the first ten elements (characters) and the last element (character) in variables. Find the number of characters in the sentence (string/array).

Solution

MATLAB Code example4.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To show examples of referencing a string as an array.
fprintf('To show examples of referencing a string as an array.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
someString = 'This is an example string.'  %Defining a given string

%------------------------- SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
%Referencing specific parts of the string
subString     = someString(1:10)     %The first 10 elements/characters
                                     %    of the string
lastCharacter = someString(end)      %The last element of the string
stringLength  = length(someString)   %Finding the length of the string
```

```
Command Window Output                                          Example 4
PURPOSE
To show examples of referencing a string as an array.

INPUTS

someString =

    'This is an example string.'

OUTPUTS

subString =

    'This is an'


lastCharacter =

    '.'


stringLength =

    26

>>
```

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Isolating a matrix element	`A()`	`A(r#,c#)`
Maximum dimension of a matrix/vector	`length()`	`length(A)`
Size of a matrix	`size()`	`size(A)`
Last element of a dimension	`end`	`A(end,1)`
Matrix referencing shorthand (whole dimension)	`:`	`A(:,1)`
Matrix referencing shorthand (partial dimension)	`a:b`	`A(1:20,1)`
Develop a zero matrix	`zeros()`	`zeros(m,n)`

Create an identity matrix	`eye()`	`eye(n)`
Develop a ones matrix	`ones()`	`ones(m,n)`
Create an arbitrary square matrix	`magic()`	`magic(n)`

Multiple Choice Quiz

(1). The MATLAB function that outputs the number of rows and columns of a matrix is

(a) `dimensions()`

(b) `mdim()`

(c) `msize()`

(d) `size()`

(2). When inputting a matrix, each new row is separated by a

(a) `:`

(b) `;`

(c) `|`

(d) `,`

(3). What is the output of the last line?

```
clc
clear
r = [1 3 -6;4 7 1;5 9 10;6 7 8];
n = length(r)
```

(a) `-6`

(b) `3`

(c) `4`

(d) `12`

(4). Given a matrix,[A], which of the following will return the element from the second row and third column?

(a) `A(3)(2)`

(b) `A(3,2)`

(c) `A(2)(3)`

(d) `A(2,3)`

(5). What is the Command Window output of the following program?

```
clc
clear
A = [1  2  3;4  5  6;7  8  9];
B = [27  8;5  0;13  2];
A(3,2)*B(1,2)
```

(a) `ans = 64`

(b) `ans = 48`

(c) `ans = 40`

(d) `ans = 30`

Problem Set

(1). Create a row and column vector each with five elements and store each of them in a variable. Then find the size and length of each vector.

(2). Create a 3×4 matrix and store it in a variable. Find its size and display the number of rows and the number of columns with a single `fprintf()` function.

(3). Create a 4×4 matrix and store it in a variable. Then find and display the product of the elements at the fourth row, second column and the second row, third column.

(4). Create an $\mathrm{n} \times \mathrm{n}$ matrix and store it in a variable. Then create a column vector from the first column of that matrix and store it in a variable. Remember not to hardcode! Your solution should work for any size matrix.

(5). Create an $\mathrm{m} \times \mathrm{n}$ matrix, where $\mathrm{m} > \mathrm{n} > 3$ and store it in a variable. Then create a row vector from the last three elements of the first row of the matrix.

Module 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.7 – How to Debug Code

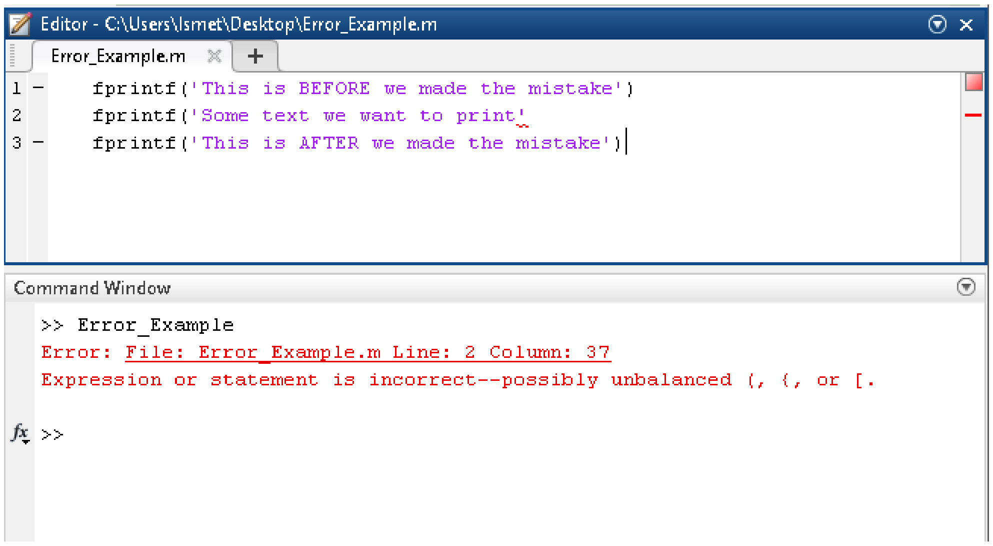
Learning Objectives

1) *identify problems in your code,*
2) *interpret errors and warnings,*
3) *debug an m-file.*

Generally, there are two problems you might have with your code: 1) it has an error and will not run, or 2) it does not do what you expect it to do. We will address the first type of problem in this lesson. For the second problem, you will have to complete the rest of the course!

What is an error?

There are two general types of messages MATLAB may return to alert you to a problem (or potential problem) it sees in your code. These are called error and warning messages, and MATLAB will display them in the Command Window (see Figure 1).



```
fprintf('This is BEFORE we made the mistake')
fprintf('Some text we want to print'
fprintf('This is AFTER we made the mistake')
```

```
>> Error_Example
Error: File: Error_Example.m Line: 2 Column: 37
Expression or statement is incorrect--possibly unbalanced (, {, or [.

>>
```

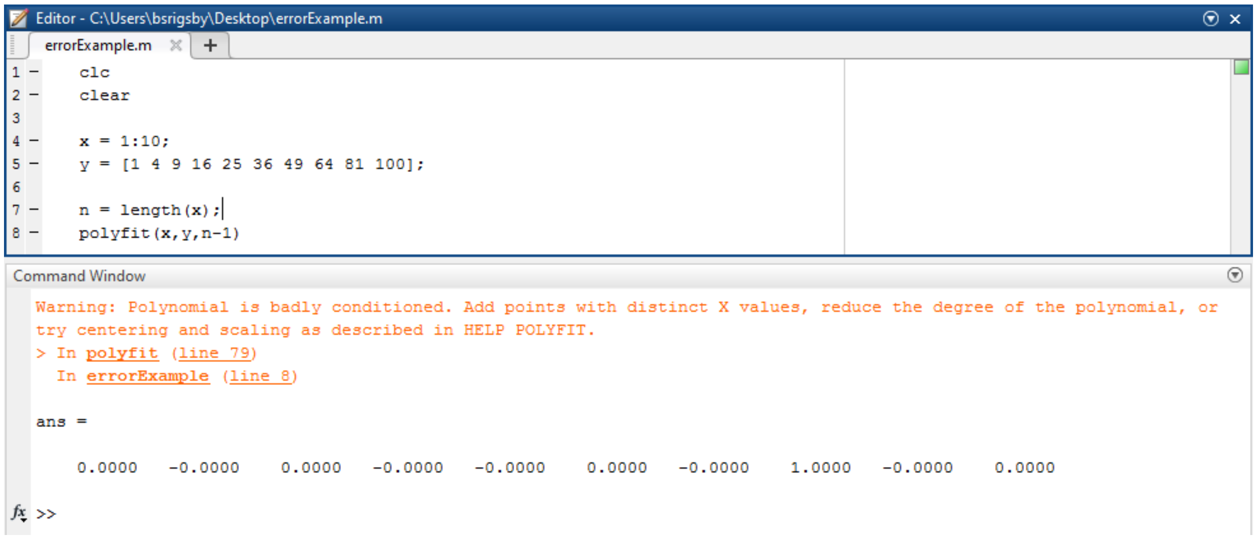
Figure 1: An example of an error in MATLAB where it has clearly identified the problem.

Error messages either will not let your program run, or will stop it before it finishes running: it depends on the error. Error messages can identify errors in syntax, problems with the way MATLAB functions are called, etc.

Figure 1 shows an example of an error in MATLAB. As you can see, our program did not execute any lines of code. We know this because the first line of code is written correctly, but it did not display any text to the Command Window. Also, note that MATLAB displays the error message in the Command Window, and tells us exactly what is wrong.

What is a warning?

A warning is for MATLAB to tell you something about your program without stopping it from running. One example of this is mathematical alerts where MATLAB has checked something behind the scenes and is telling you to be careful (see Figure 2). This is done by the programmers who wrote that MATLAB function to alert you to a specific mathematical problem rather than a programming one. Another example is an outdated or "depreciated" code warning. Computer languages decide to remove functions sometimes and will display "depreciation warnings" that function/syntax will not be supported in future releases of the software (MATLAB in this case).



Editor - C:\Users\bsrigsby\Desktop\errorExample.m

errorExample.m

```
clc
clear

x = 1:10;
y = [1 4 9 16 25 36 49 64 81 100];

n = length(x);
polyfit(x,y,n-1)
```

Command Window

```
Warning: Polynomial is badly conditioned. Add points with distinct X values, reduce the degree of the polynomial, or
try centering and scaling as described in HELP POLYFIT.
> In polyfit (line 79)
  In errorExample (line 8)

ans =

    0.0000   -0.0000    0.0000   -0.0000   -0.0000    0.0000   -0.0000    1.0000   -0.0000    0.0000

>>
```

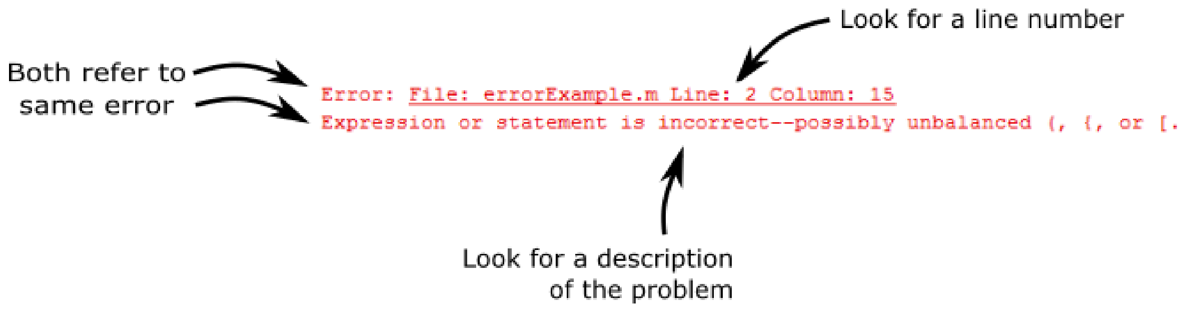
Figure 2: An example of a warning in MATLAB where it has identified a potential mathematical problem.

How can I solve the problem with my code?

One should use a systematic approach to fix, or "debug" a programming problem. The following sub-sections review a general step by step process to identify why your code is broken. There are a variety of errors that can occur in MATLAB; however, the following sections will give you a general way of approach to tackle most errors that you encounter.

Most of the time, if you make an error in the syntax of your program, MATLAB will return an error. Typically, error messages contain at least two types of helpful information. As you can see in Figure 1, it provides a line reference where it believes the problem is occurring (this is often the correct line, but not always). The error message also usually contains some helpful information about the problem, and this information has gotten more and more detailed in newer versions of MATLAB.

✓ ***Important Note:*** Always read the error message in full!



```
Error: File: errorExample.m Line: 2 Column: 15
Expression or statement is incorrect--possibly unbalanced (, {, or [.
```

Figure 3: Example of how to extract information from an error message.

Once you have gleaned all the information from an error message, then you can move on to further identify and/or fix the issue. There is no exact procedure to follow when debugging, but here are some general steps to try and remember:

1. Check for common causes of errors (often identified in the error message)
    a. Line breaks (hitting Enter/Return) in the middle of a function
    b. Misspelling (case sensitive) of any used command and function
        i. Track usage of variable names in program expressions
        ii. Ensure that no single variable has more than one name
2. Check code syntax
    a. Read the documentation relevant to the commands/functions you are using
    b. Check function input order placement (Syntax)
        i. Ensure that each function has the correct number and order of inputs
        ii. Ensure each input is of the correct data type (integer, string, vector, etc.)
3. Think about what is happening in the program!
    a. Unsuppress variables to see what values they are taking on.
    b. To quickly see all the uses of a variable name in an m-file, click anywhere in the variable name and wait a moment. See Figure 4.
    c. It can help to hand-write out the steps the computer will go through. As a rule of thumb, if you do not know how to do something on paper, you do not know how to do it in your program.
    d. Check program flowchart, and/or I/O diagram, and/or pseudocode (See Lessons 6.1 and 6.2)

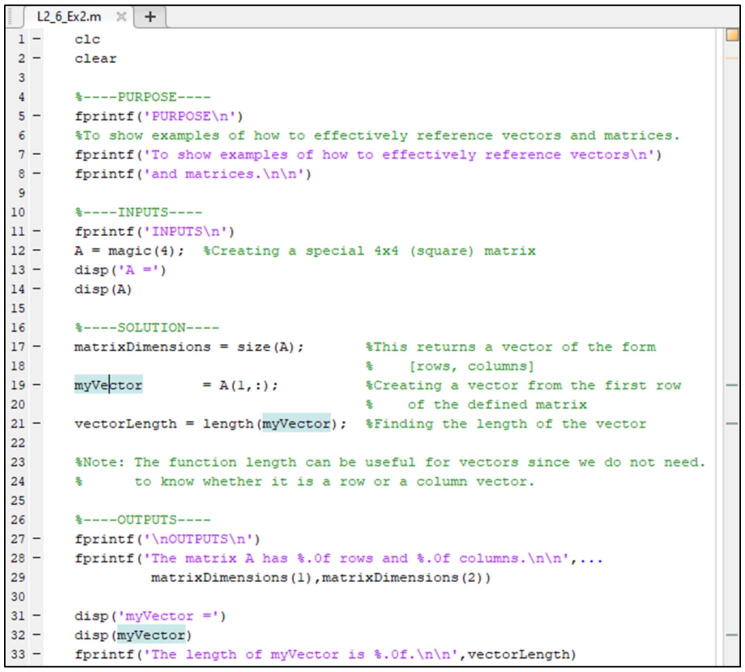
i. This is more helpful when the program does not have errors and still does not return the expected output.

4. Check custom, user-defined functions (See Lesson 7.1)

   a. Repeat Steps 1 – 8 for any user-defined functions

   b. If there is an error originating from a user-defined function, you can see that in the error message. The m-file where the error occurred is listed in the error message (see Figure 3).

Figure 4 shows an example of the instant find feature of MATLAB. This allows you to click on any variable and have all the instances marked and highlighted. In this case, the variable `myVector` has been clicked on at line 19 (although you can click on any instance of the variable), and MATLAB has highlighted the other two instances in blue (seen in Figure 4 as grey) on lines 21 and 32.



```
L2_6_Ex2.m
 1 -   clc
 2 -   clear
 3
 4     %----PURPOSE----
 5 -   fprintf('PURPOSE\n')
 6     %To show examples of how to effectively reference vectors and matrices.
 7 -   fprintf('To show examples of how to effectively reference vectors\n')
 8 -   fprintf('and matrices.\n\n')
 9
10     %----INPUTS----
11 -   fprintf('INPUTS\n')
12 -   A = magic(4);  %Creating a special 4x4 (square) matrix
13 -   disp('A =')
14 -   disp(A)
15
16     %----SOLUTION----
17 -   matrixDimensions = size(A);       %This returns a vector of the form
18                                       %    [rows, columns]
19 -   myVector        = A(1,:);         %Creating a vector from the first row
20                                       %    of the defined matrix
21 -   vectorLength = length(myVector);  %Finding the length of the vector
22
23     %Note: The function length can be useful for vectors since we do not need.
24     %      to know whether it is a row or a column vector.
25
26     %----OUTPUTS----
27 -   fprintf('\nOUTPUTS\n')
28 -   fprintf('The matrix A has %.0f rows and %.0f columns.\n\n',...
29              matrixDimensions(1),matrixDimensions(2))
30
31 -   disp('myVector =')
32 -   disp(myVector)
33 -   fprintf('The length of myVector is %.0f.\n\n',vectorLength)
```

Figure 4: m-file showing the find feature.

One can see `myVector` highlighted in three different places in Figure 4. If your program is longer, you can see that there are grey dashes on the right side of the Editor that show the relative location within the m-file. This makes it a lot easier and faster to check all the different usages of a variable name.

Can I pause my program part of the way through?

There are two simple ways to pause your program partway through a run. These are breakpoints and the “Pause” button in the Editor tab (Figure 5).

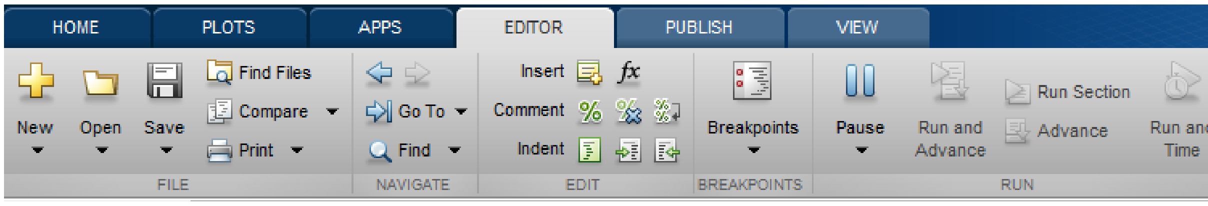


Figure 5: Showing the “Pause” button in the Editor tab.

The “Pause” button, once clicked, simply pauses the program wherever it is currently in the execution of the m-file. For example, if Pause is clicked as MATLAB executes line 43, it will pause execution at line 43. If execution is resumed by the programmer (Figure 6), the m-file will continue from line 43.

Breakpoints are a great tool for quickly debugging a program. They function similar to the way the Pause button works except the location where they pause the program execution is precise. Typically, you should favor breakpoints over the Pause button due to the precision of breakpoints.

Figure 7 shows several breakpoints placed in an m-file. Notice the breakpoint circles on lines 17 and 27. Although we used two breakpoints in Figure 7, you can use as many as you like. Placing a breakpoint is done by using the mouse cursor to click in the space to the right of the m-file line number. A breakpoint is denoted with a circle (filled with grey or red color) and can be removed by clicking inside the circle.

When the program execution has been paused, it can be resumed or stopped. The user interface (UI) buttons for this in MATLAB are shown in Figure 6. If “Continue” is selected, the program (m-file) continues as it normally would.

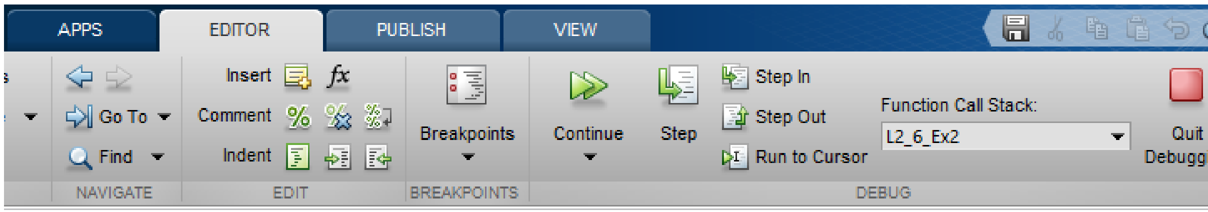
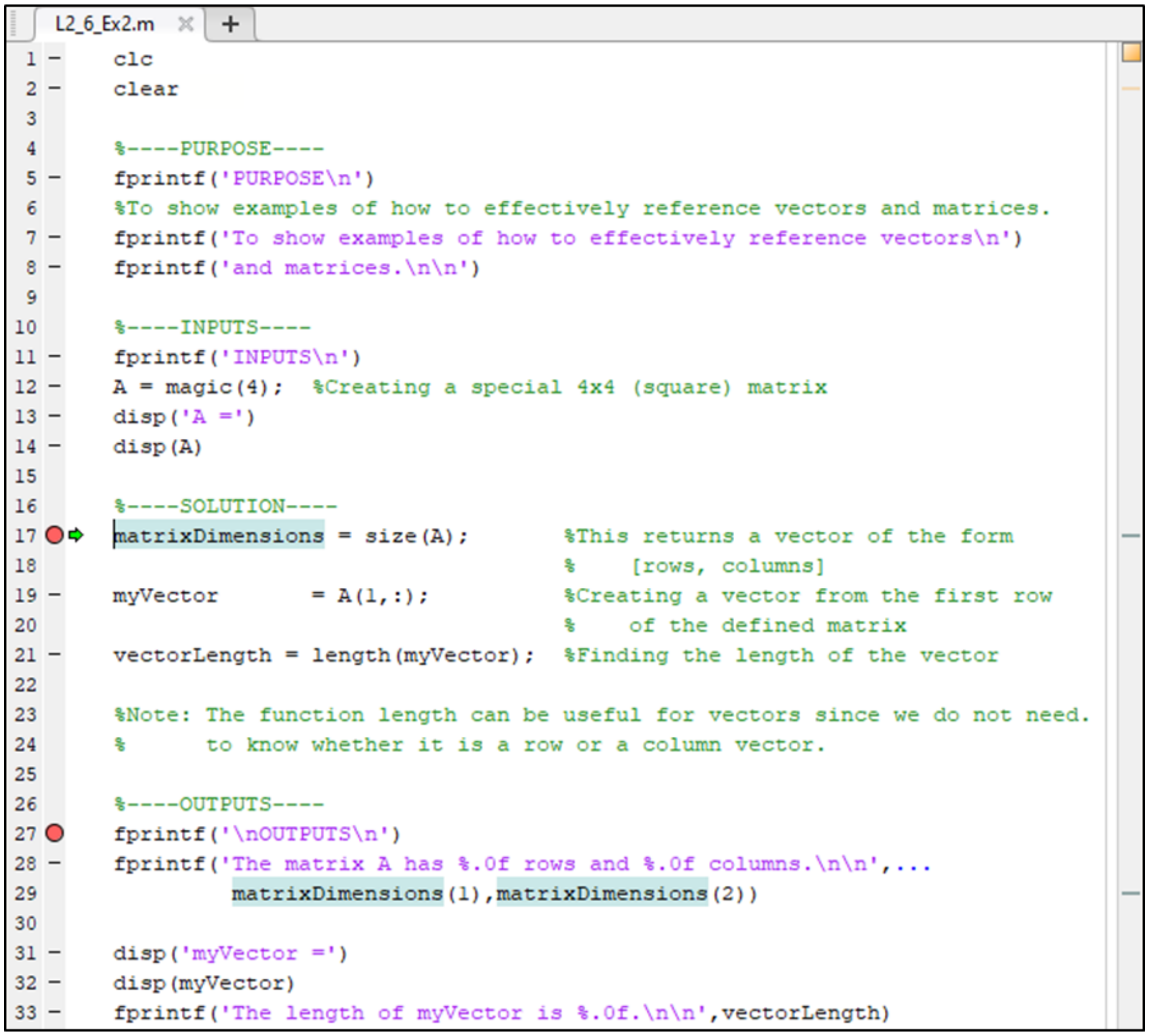


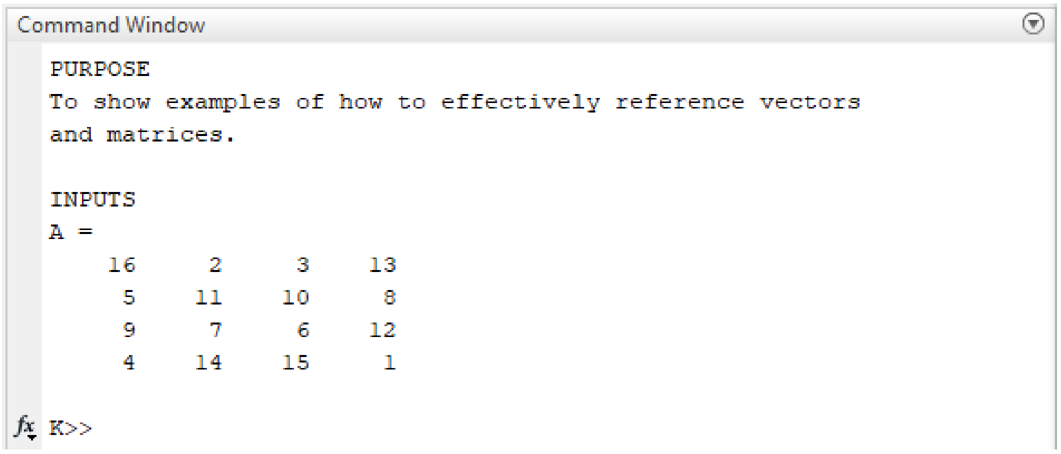
Figure 6: Run options when the program has paused.

When an m-file is first executed (use normal execution; i.e., “Run”) with breakpoints, only the segment of code prior to the first breakpoint will run. The m-file will then sequentially execute each segment of code to the next breakpoint for each single execution. After each execution, a green arrow will appear that points to the next executable line of code. Figure 7 shows the indicator arrow (green in MATLAB) and a program m-file which has been partially executed using breakpoints.



```
clc
clear

%----PURPOSE----
fprintf('PURPOSE\n')
%To show examples of how to effectively reference vectors and matrices.
fprintf('To show examples of how to effectively reference vectors\n')
fprintf('and matrices.\n\n')

%----INPUTS----
fprintf('INPUTS\n')
A = magic(4);  %Creating a special 4x4 (square) matrix
disp('A =')
disp(A)

%----SOLUTION----
matrixDimensions = size(A);        %This returns a vector of the form
                                   %    [rows, columns]
myVector        = A(1,:);          %Creating a vector from the first row
                                   %    of the defined matrix
vectorLength = length(myVector);   %Finding the length of the vector

%Note: The function length can be useful for vectors since we do not need.
%      to know whether it is a row or a column vector.

%----OUTPUTS----
fprintf('\nOUTPUTS\n')
fprintf('The matrix A has %.0f rows and %.0f columns.\n\n',...
         matrixDimensions(1),matrixDimensions(2))

disp('myVector =')
disp(myVector)
fprintf('The length of myVector is %.0f.\n\n',vectorLength)
```

Figure 7: Breakpoints placed on an m-file.



```
PURPOSE
To show examples of how to effectively reference vectors
and matrices.

INPUTS
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

K>>
```

Figure 8: Command Window output for code in Figure 6 at the first breakpoint (line 17).

Clicking Run on the m-file shown in Figure 7 will execute lines 1 – 16. It will pause before executing line 17 (and beyond) because we have placed a breakpoint there. Clicking Continue will execute the code from line 17-26. It will pause before line 27 because we have placed a breakpoint there. Finally, clicking Continue the last time will run the rest of the m-file as there are no more breakpoints. Take note of the outputs shown in Figure 8. These are the outputs before the first breakpoint.

What if I cannot find the exact place of the error?

If you cannot pinpoint the error from an error message or some other information you have gathered, you should try to isolate the problem through other methods. Commenting (%) the part of the code that is nor relevant to the error is a good way to track down the source of the error. You can also make smart use of breakpoints. Remember, MATLAB reads lines of code sequentially, so you should start at the top and work down or vice versa. Unsuppressing (removing the semicolon ";" at the end of a line of code) shows the value of the variable in the Command Window.

Multiple Choice Quiz

(1). An error given by a MATLAB program means

(a) the program will continue running, but something is wrong

(b) something is wrong, and the program will not continue to run

(c) MATLAB crashed and you need to re-run the program

(d) you must manually save the program before running it

(2). A good practice while debugging a program is

(a) unsuppressing a Command Window output

(b) suppressing a Command Window output

(c) removing comments

(d) removing `fprintf()` statements

(3). The "Pause" button, when pressed at a given line, does the following

(a) resumes the program from that line

(b) pauses the program at that line

(c) stops the whole program

(d) restarts the program

(4). The number of breakpoints you can use in a program is

(a) one

(b) two

(c) one for each line of code

(d) ten

(5). A warning given by a MATLAB program implies

(a) the program has an error

(b) the program has potential issues that need to be addressed

(c) the end-user of the program has little to worry about the output of the program

(d) the program will not run

Problem Set

Note that you should not expect an error for every "problem". Errors are returned by MATLAB when there is a problem with syntax, but not necessarily when there is a problem with your solution in general.

(1). Find the problem(s) in the following code and fix them. Then write one paragraph describing the process you used to debug the code. How did you identify the problem(s) and find the solution(s)?

```
clc
clear

height = 7.8;
length = 15;
area = height*length;
fprintf('The area of the rectangle is %s.\n',Area)
```

(2). Find the problem(s) in the following code and fix them. Then write one paragraph describing what the problem(s) were. Reference the documentation as well.

```
clc
clear

A = [3.4 5.55 1;0.4 2;100 6 45.3]
vector = 1:2:10;
disp(vector)
```

(3). Find the problem(s) in the following code and fix them. Then write one paragraph describing what the problem(s) were. Reference the documentation as well.

```
clc
clear

constant = 5;
matrix = [12 0 43;constant 33 1.2
disp(matrix)
```

(4). Find the problem(s) in the following code and fix them. Then write one paragraph describing what the problem(s) were. Reference the documentation as well.

```
clc
clear

B = [2 3 4;
    2 1 45;
    9 3 6];

sumElem = B(1,1) + B(2,4) +  B(3,3);
fprintf('The sum of three elements is %g.\n'sumElem)
```

(5). Find the problem(s) in the following code and fix them. Then write one paragraph describing what the problem(s) were. Reference the documentation as well.

```
clc
clear

a = 2.1;
1stSensor = 12;
fprintf('The first sensor value is %.2fV.\n\n',1stSensor)
correctedVolt = b*1stSensor
b = 1.003;
```

Module 3: PLOTTING

Lesson 3.1 – Plots and Figures

Learning Objectives

1) *display two-dimensional plots,*
2) *display several plots on one graph,*
3) *display logarithmic plots.*

How can I visualize (plot) data in MATLAB?

Making a plot is vital for analyzing and understanding data as well as relaying information to your colleagues. Plots are commonly used in books, reports, and many other engineering documents. This lesson will cover the basic concepts of developing plots using MATLAB.

Ask yourself, "What information do I want to plot?" MATLAB has many methods for developing a plot, but we will focus on the plotting of lines in this lesson. The plot will require two vectors. One of these vectors is the horizontal (abscissa) axis values and the other vector is for the vertical (ordinate) axis values.

How do I plot data pairs (points) in MATLAB?

Function: Plotting sets of 2D data pairs.
`plot()`

Function Inputs:
x data points (enter as vector), `x`
y data points (enter as vector), `y`

Example Usage:
`plot(x,y)`

Once the m-file has been run, a new window will appear that contains the data points on a plot. This figure will be titled as Figure 1, and that is your plot.

Example 1

Using plotting features of MATLAB, plot the following (x, y) data pairs on a linear graph. The data pairs to plot are: (0,0), (1,1), (2,4), (3,9), (4,16).

Solution

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
%To plot a 2D line (data set)

%----------------------------- INPUTS ------------------------------
x = [0 1 2 3 4];    %Defining x points
y = [0 1 4 9 16];   %Defining y points

%------------------------ SOLUTION/OUTPUTS -------------------------
plot(x,y)           %Plotting x and y points
```

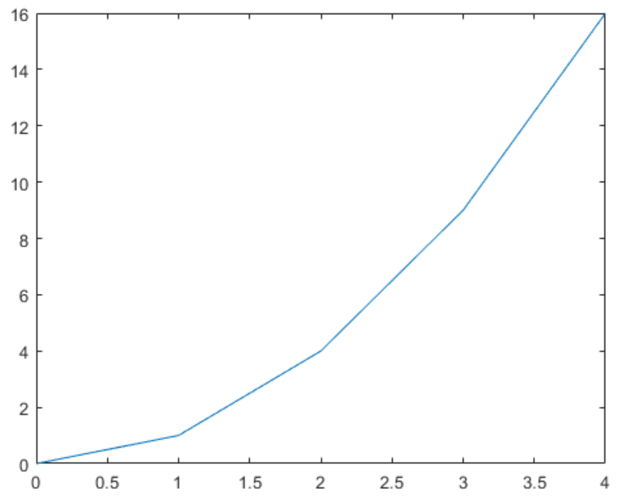


Figure 1: The figure output by the code in Example 1.

As discussed in Lesson 2.6, when inputting data points make sure you use brackets to establish a vector and separate each data point with a space. Note that we have turned the data pairs into two vectors. The names of these two vectors do not have to be `x` and `y` as they can be named as any legal variable names. Also, note that the plot simply joins the given data points with straight lines.

How do I plot a function in MATLAB?

When MATLAB is plotting a function, the procedure is basically doing the same thing as when plotting discrete points as shown in Example 1. The only difference is in the creation of the points that we want to plot. For a continuous function, we use the values from an `x` vector to build the `y` vector by plugging individual values of `x` into the desired function. When developing the domain of `x` values, the `interval` is the space between each point. The smaller the interval, the resolution of the plot and the computational time to create the plot both increase.

Example 2

Plot the given function using MATLAB plotting functions. To generate the y values from the function, use x values from 0 to 4 with an interval between each point of 0.1.

Function: $\mathrm{y} = 2\mathrm{x} + 1$

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
%To plot a 2D line representing a function

%------------------------------ INPUTS -------------------------------
x = 0:0.1:4;     %Creating a vector of x points with an interval of 0.1
y = 2*x+1;       %Creating a vector of y points by plugging in the x
                 %     points to the function

%------------------------- SOLUTION/OUTPUTS -------------------------
plot(x,y)        %Plotting x and y points
```

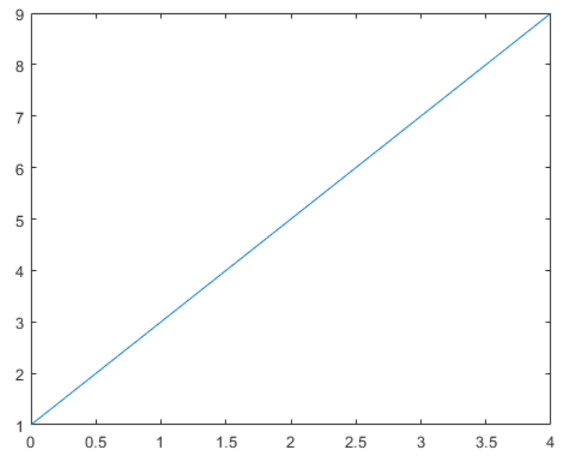


Figure 2: The figure output by the code in Example 2.

When generating the x vector, you should take note at the notation used (x_1 (first data point): interval: x_n (last data point)). Having a colon (:) to separate the starting point, the interval, and the final point is not just used when plotting, it is also used when generating vectors in MATLAB. You can see this in Example 1, and we will use this notation again soon.

What is the difference between a figure and a plot?

A "figure" is the window containing a "plot". A plot is a specific graphic that is displayed in the figure. You can see they are called separately in Example 1 where we first open a figure and then print a plot to it. Therefore, a figure can exist without a plot, but not vice versa (see Figure 3).

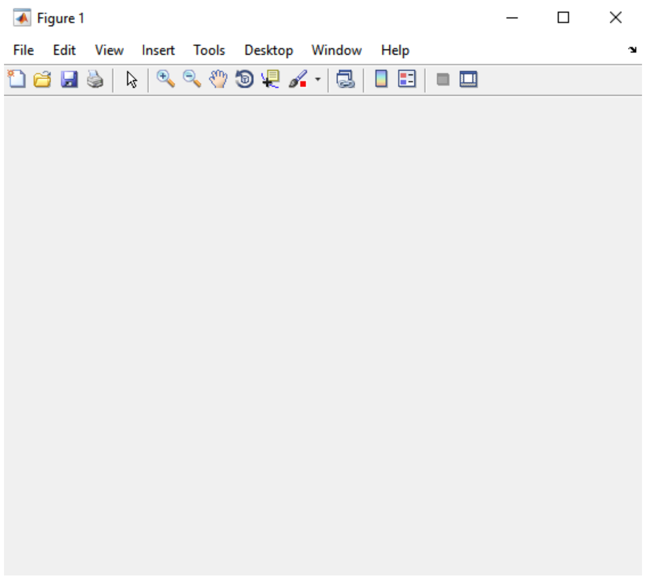


Figure 3: A blank MATLAB figure.

How can I enter nonlinear functions for plotting?

Many nonlinear functions that you want to plot will require the use of an array operator. An array operator simply means you must use the dot (.) character before the mathematical operator (e.g., `A.*B`). We will deliberately shorten our explanation of this operator for now. Just know that it comes from linear algebra rules, which we will cover in detail in Lesson 4.6.

To makes things simpler for this lesson, we provide specific guidelines for when to use an array operator in this lesson. Although the following guidelines are *not true in general*, they are true for the examples and exercises seen in this lesson. They are also generally true when plotting, although you should carefully review the detailed explanations in Lesson 4.6.

Use an array operator when:

1. two vectors are multiplied together. Example: `vecA.*vecB`
2. a vector is in the denominator. Example: `2./vec`
3. a vector is raised to a power. Example: `vec.^2`

Do not use an array operator when:

1. a vector is an input for trig or log functions. Example: `sin(vec)`
2. a vector is multiplied by a scalar. Example: `2*vec`

What are some possible errors with plotting?

While plotting, the following error message may pop up in the Command Window:

```
Error using plot
Vectors must be the same length.
```

This error occurs because MATLAB needs to have the same number of elements in both the x and y vectors to develop a plot. This is also true of all 2D plots in general. You can use the length() function to quickly check the lengths of the vectors you are trying to plot.

Another common error:

```
Error using *
Inner matrix dimensions must agree.
```

The cause of this error often lies in not using the array (.) operator when plotting nonlinear functions. Make sure you follow the guidelines outlined above.

You may also have this error appear:

```
Undefined function or variable...
```

If this message appears, most likely you have not been consistent in your naming of your variables. Make sure that your vector names correspond to the plot() function inputs; remember that MATLAB syntax (functions and variables) is case sensitive.

How do I show multiple data sets on the same plot?

Often, you will want to put more than one plot on the same figure. The command `hold on` can be used when you want to make multiple plots on the same figure. It should always be placed right after the first plotting function. In Example 3, you can see two `plot()` functions are used along with the `hold on`. Because the `plot()` function is used twice, MATLAB would "normally" default to generating two figures; however, using the `hold` command tells MATLAB to place these two sets of data on the same figure. The `hold off` is used when you have completed all the plotting calls for that figure. You can place as many data sets on the same figure as you want. Just make a new `plot()` function for each new data set. Finally, note that `hold` can be used with most other plotting functions: not just `plot()`.

Example 3

Plot the given data pairs and the function on the same plot using MATLAB plotting functions. To generate the y values from the function, use x values from 0 to 4 with an interval between each point of 0.1.

Data Pairs: (0, 0), (1, 1), (2, 4), (3, 9), (4, 16)
Function: $\mathrm{y} = 2\mathrm{x} + 1$

Solution

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
%To plot 2D lines representing multiple data sets

%------------------------------ INPUTS ------------------------------
x = [0 1 2 3 4];      %Defining x points
y = [0 1 4 9 16];     %Defining y points

xFunc = 0:0.1:4;      %Creating a vector of x points with 0.1 interval
yFunc = 2*xFunc + 1;  %Creating a vector of y points by plugging in x
                      %    points to the function

%------------------------ SOLUTION/OUTPUTS -------------------------
plot(x,y)          %Plotting x and y points for data set vectors
hold on            %Telling MATLAB to place new plots on the same figure
plot(xFunc,yFunc)  %Plotting x and y points for function vectors
hold off
```

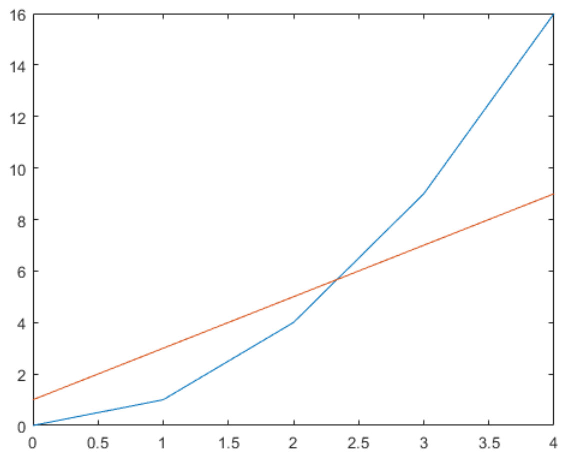


Figure 4: The figure output by the code in Example 3.

What are some other types of plots that MATLAB can generate?

An entire course can be devoted to making plots in MATLAB. Many different parameters can be controlled, animated, and viewed in different dimensions. Also, contours can be added, and labels and highlights can be placed in the graph. In this section, we cover one area of this extensive topic: the log-log and semi-log plots.

The loglog plot has a log-scale on both the x- and y-axis.

Function: Creating a log-log plot
`loglog()`

Function Inputs:
Domain of x values (vector), `x` Values of y at x values generated using the function, `y`

Example Usage:
`loglog(x,y)`

A semi-log plot is different from the log-log plot because it has a log-scale on only on one of its axes; its x or y axis.

Function: Creating a semi-log plot
`semilogx()` <u>or</u> `semilogy()`

Function Inputs:
Domain of x values (vector), `x`
Values of y at x values generated using a function of `x`, `y`

Example Usage:
`semilogx(x,y)` <u>or</u> `semilogy(x,y)`

Example 4

Develop a linear-log plot with the logarithmic axis being the horizontal axis. The function to plot is $y = log(x)$. Note the MATLAB function for this is `log()`. Use the function provided and a range of independent values going from 1 to 3000 with an interval of 0.1. Be sure to suppress the vertical and horizontal vectors as they contain thousands of elements.

Solution

MATLAB Code example4.m

```
clc
clear
close all

%----------------------------- PURPOSE -----------------------------
%To plot a 2D line on with linear-log axes
```

MATLAB Code (continued) example4.m

```
%----------------------------- INPUTS ------------------------------
x = 1:0.1:3000;    %Creating a vector of x points with an interval of 0.1
y = log(x);        %Creating a vector of y points by plugging in x points
                   %    to the function

%------------------------ SOLUTION/OUTPUTS -------------------------
semilogx(x,y)      %Plotting x and y points for data set vectors
```

One can see that the `semilogx()` function is used to create a log-scale on the horizontal or x axis of the figure.

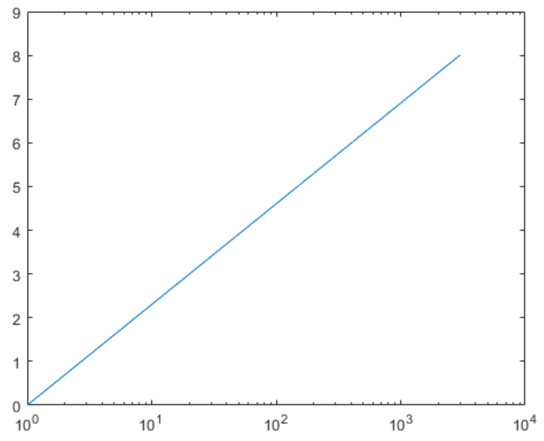


Figure 5: The figure output by the code in Example 4. Notice the x-axis is logarithmic.

How do I plot on more than one figure in the same m-file?

You have probably noticed that if you try to plot more than one figure using the same m-file, only the last figure appears; that is, MATLAB overwrites the preceding figures. The solution to this problem is simple: use the `figure` command.

To number a figure, place the command prior to the `plot` function. For example, to plot two different graphs, one would type,

```
figure
plot(x,y)
…
figure
plot(xp,yp)
```

MATLAB will open a new figure window each time the `figure` command is executed. To display different figures in one window, the function `subplot()` is used (This function is not discussed in the book. You can search MATLAB help for `subplot()` to learn more about this function).

What is the close command?

The `close` command allows one to close one or more open MATLAB windows. The `close all` command will close all open figure windows. This will close all MATLAB figure windows that are open when it executes. If one has numbered the figures in their m-file, MATLAB will continue to add to the figures during each run of the m-file. This is similar to how the Command Window will continue to display the output(s) from all previous runs if the `clc` command is not used. This can become very cumbersome, so it is usually a good idea to place this command at the beginning of your program (following the `clc` and `clear` commands). Another option to clear figures is the `clf` command, which stands for clear all figures. The difference here is that it will *clear* the figure windows, but it will not *close* the windows.

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Open a new blank MATLAB figure window	`figure`	`figure`
Plot 2D data points on linear axes	`plot()`	`plot(x,y)`
Plot 2D data points on logarithmic axes	`loglog()`	`loglog(x,y)`
Plot 2D data points w/logarithmic x-axis	`semilogx()`	`semilogx(x,y)`
Plot 2D data points w/logarithmic y-axis	`semilogy()`	`semilogy(x,y)`
Close all open MATLAB figure windows	`close`	`close all`

Multiple Choice Quiz

(1). The MATLAB function to make a plot is

(a) `figure()`

(b) `fig()`

(c) `plot()`

(d) `pplot()`

(2). The standard input form for the MATLAB function `semilogx()` for plotting y vs. x data is

(a) `semilogx(log(x),y)`

(b) `semilogx(x,y)`

(c) `semilogx(log(x),log(y))`

(d) `semilogx(x,log(y))`

(3). The interval between the points in the array `xx = 3:0.1:20` is

(a) 0.1

(b) 3

(c) 10

(d) 20

(4). To plot y=2x from x=3 to 7, the code would be

(a)
```
x = 3:0.1:7;
y = 2*x;
plot(x,y)
```

(b)
```
x = 3:0.1:7;
y = 2*x;
plot(y,x)
```

(c)
```
x = 7:0.1:3;
y = x*x;
plot(x,y)
```

(d)
```
x = 7:-0.1:3;
y = 2*x;
plot(y,x)
```

(5). The vector `xp = 3:0.5:9` yields ____________ elements?

(a) 6

(b) 7

(c) 12

(d) 13

Problem Set

(1). For the duration of one year, the total sales profit for a turbine company is recorded. At the end of each month, the total profit (in millions of dollars) for that month is calculated and stored in MATLAB as a vector, profit.

$$\text{profit} = [0.3 \;\; 0.45 \;\; 2.1 \;\; 1.4 \;\; 1.12 \;\; 3.2 \;\; 2.3 \;\; 1.23 \;\; 0.76 \;\; 0.97 \;\; 1.2 \;\; 0.78]$$

Using MATLAB graphing features, plot the profit vs. time (months). Also, using the `fprintf()` function, output the total profit (in millions) for the year.

(2). Plot the given discrete data set (as green points) and function (as a blue dash-dot line) on the same plot. Use the minimum and maximum of the x values in the discrete data set as the plotted domain of the function with an interval of 0.2.

Data points: (2.2, 6.4), (2.8, 11), (3.0, 13), (5.0, 35), (6.1, 51), (8.0, 83)
Function: y = 5x − 3

(3). A rocket is horizontally strapped to the top of a sled and ignited. The position of this contraption is given as a function of time (sec) by

$$s(t) = \frac{3}{50}t^3 + \frac{7}{30}t^2 - 5t \quad \text{(ft)}$$

Plot the position of the sled in MATLAB from 0 to 60 sec. Remember to follow the guidelines given in this lesson for raising a vector to a power when plotting.

(4). The kinetic energy of a system can be modeled as

$\text{KE} = \frac{1}{2}m\left|v^2\right|,$

where
KE is the kinetic energy $(N \cdot m)$
m is the system mass (kg)
v is the velocity $(\frac{m}{s})$

Plot the kinetic energy of a system vs. velocity in MATLAB. Use values of mass as 2 kg and take the range of velocity as 0 to 30 m/s. Remember to follow the guidelines given in this lesson for raising a vector to a power when plotting.

(5). The velocity of a rocket is given by,

$v(t) = 2000\ln\left[\frac{14\times10^4}{14\times10^4-2100t}\right] - 9.8t, \quad 0 \le t \le 30$

where v is given in m/s and t is given in seconds. Plot the velocity of the rocket as a function of time from 0 to 30 seconds. Use a loglog plot. The function for finding a natural log is `log()` (see Lesson 4.1 or use MATLAB documentation). Remember to follow the guidelines given in this lesson for a vector in the denominator when plotting.

Module 3: PLOTTING

Lesson 3.2 – Plot Formatting

Learning Objectives

1) *add axis labels to MATLAB plots,*

2) *add a legend to MATLAB plots,*

3) *add a title to MATLAB plots,*

4) *add special characters to text in MATLAB plots,*

5) *improve the overall look of MATLAB plots.*

How can my MATLAB graph look nicer?

As you can tell from Lesson 3.1, developing graphs can prove to be important for interpreting data. The ability to make those graphs easier to read and aesthetically pleasing is equally important. In this lesson, you will learn several techniques to make a MATLAB plot more readable and easier to follow. This section will cover the functions and commands on how to make a legend, title, axis labels, and place a grid onto a plot. Also covered are the use of special fonts and characters and how to change the axis markings.

What are some terms I should know for plots?

Figure 1 shows the MATLAB naming convention for plotting. Most of the nomenclature is common sense and similar to other software, but it is important to know to understand how to change the different properties.

In this lesson, we will cover the two most commonly used groups of properties that define how plotted graphics look in MATLAB. Those are as follows:

- Line Properties define chart line appearance and behavior: for example, the line style and thickness.
- Axis Properties define axes appearance and behavior: for example, axis limits, title, and legend.

The properties are not mandatory as you could see from the last lesson. In other words, you could make a plot without MATLAB requiring you to have a title, line width, axis labels, etc.

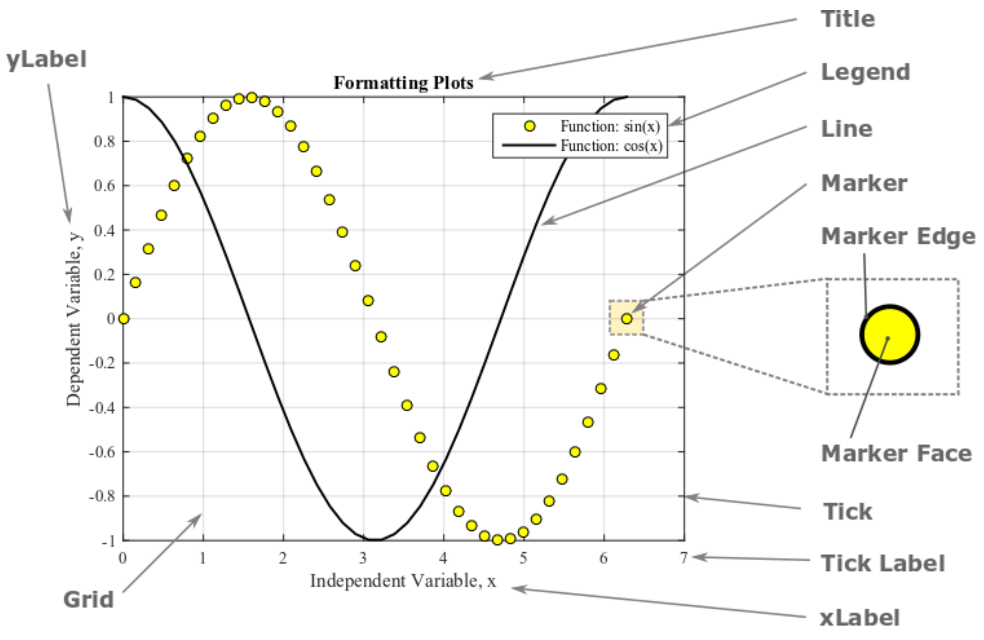


Figure 1: A MATLAB plot with common plot properties annotated.

How can I change the color and style of lines and markers on a plot?

MATLAB provides many different options to change how your plots look. You can see some of the common options in Tables 1, 2, and 3 below. Example 1 uses some of these options to customize how a figure appears.

Table 1: Various marker plotting styles.

Desired Style	**Syntax**	**Example Usage**
Circle	`'o'`	`plot(x,y,'o')`
Square	`'s'`	`plot(x,y,'s')`
Asterisk	`'*'`	`plot(x,y,'*')`
Cross	`'+'`	`plot(x,y,'+')`
Small point	`'.'`	`plot(x,y,'.')`
Diamond	`'d'`	`plot(x,y,'d')`
Five-pointed star	`'p'`	`plot(x,y,'p')`

Table 2 shows different line styles used to represent the function in the generated plot. To change the color of a line or data point, use the parameters given in Table 3. Note that these color and line style options can be used in the same parameter input to the `plot()` function. For example, we can instruct MATLAB to make the plot a dotted red line with `plot(x,y,'r:')`.

Table 2: Various line plotting styles.

Desired Line Style	**Syntax**	**Example Usage**
solid line (default)	`'-'`	`plot(x,y,'-')`
dashed line	`'-'`	`plot(x,y,'-')`
dotted line	`':'`	`plot(x,y,':')`
dash-dot line	`'-.'`	`plot(x,y,'-.')`

Table 3: Various color options for data points and lines.

Desired Color	**Syntax**	**Example Usage**
blue (default)	`'b'`	`plot(x,y,'b')`
red	`'r'`	`plot(x,y,'r')`
black	`'k'`	`plot(x,y,'k')`
yellow	`'y'`	`plot(x,y,'y')`
magenta	`'m'`	`plot(x,y,'m')`
green	`'g'`	`plot(x,y,'g')`

Desired Color	**Syntax**	**Example Usage**
cyan	`'c'`	`plot(x,y,'c')`

How can I make the function and points on the graph look nicer?

There are many ways to display the desired function(s) and/or data points on a graph. To name a few, modifications include the change of the following - color, size, shape, line type, and outline of both the points and function. Typing `doc plot` in the Command Window will yield tables of information and codes that can be used to modify your graph.

Line Parameter: Line width of a curve (function)
`'LineWidth'`

Parameter Value:
Any positive integer – the larger the number the thicker the line width

Example Usage:
`plot(x,y,'LineWidth',2)` (read more about code placement below)

Line Parameter: Size of a data point symbol
`'MarkerSize'`

Parameter Value:
Any positive integer – the larger the number the larger the marker size

Example Usage:
`plot(x,y,'MarkerSize',6)` (read more about code placement below)

Both the `LineWidth` and `MarkerSize` parameters must be used in conjunction with the `plot()` function as they are parameters of this function; therefore, they must go inside the `plot()` function. To illustrate this, Example 1 shows both the m-file and generated figure using these parameters.

Example 1

Plot the function $y = 2.1x + 4.4$ and the following data set on the same plot. Data pairs to plot are: (0, 4.12), (2, 8.6), (4, 11.5), (5, 15.3), (7, 18.0), (8.5, 21.25). Plot the function within the domain of 0 to 10 with an interval between the points of 0.1. Use blue points (markers) for the data set with a specified marker size of 6 and a red dotted line for the function with a line width of 2.

Include a legend, title, and axis labels on the plot. Use bold font for the title and italicize the axis labels.

Solution

MATLAB Code example1.m

```
clc
clear
close all

%----------------------------- PURPOSE -----------------------------
%To put multiple data sets on the same plot with professional formatting

%----------------------------- INPUTS ------------------------------
x = [0     2    4     5     7     8.5];     %Defining x points
y = [4.12  8.6  11.5  15.3  18.0  21.25];   %Defining y points

xFunc = 0:0.1:10;          %Generating the domain/independent variable
                           %    values for the function
yFunc = 2.1*xFunc + 4.4;   %Generating the range/dependent variable
                           %    values for the function

%------------------------ SOLUTION/OUTPUTS -------------------------
plot(x,y,'bo','MarkerSize',6)          %Plotting x and y points for data
                                       %    set vectors
hold on                                %Telling MATLAB to place new plots
                                       %    on the same figure
plot(xFunc,yFunc,'r:','LineWidth',2)   %Plotting x and y points for
                                       %    function vectors
hold off
```

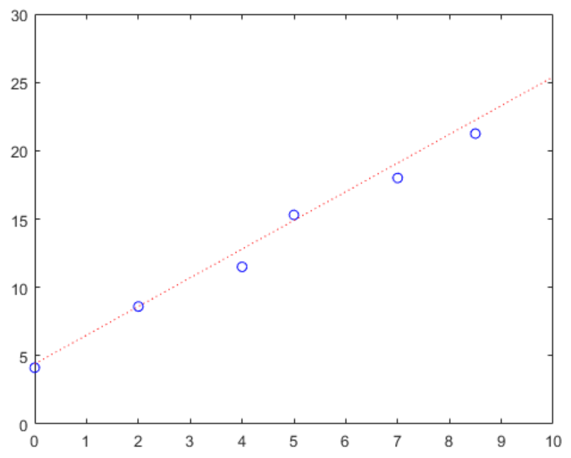


Figure 2: The figure output by the code in Example 1.

How can I put a title and axis labels on my plot?

Whenever you are making a plot, you should always use a title and axis labels. Even if it is the world's simplest plot, these things are important. MATLAB contains several preprogrammed functions that allow the user to add a figure title and axis labels to a graph.

Example 2 shows an m-file with the corresponding figure using the `title()`,`xlabel()`, and `ylabel()` functions.

How can I add a legend to my plot?

The function to add a legend to a plot is `legend()`. In the m-file, the `legend()` function must be placed after the last plotting call. The order in which the descriptions *should* appear in the legend is the same as the order in which the functions/points are plotted. When making a legend, double-check that the descriptions match with what MATLAB is plotting. MATLAB will automatically place the line style of the function/point with the description provided by the user for each object in the legend.

Example 2 shows an example m-file of how to use a legend in a MATLAB-generated figure. Notice the use of the `'location'` parameter to manually set the location of the legend on the plot. The parameter 'NW' (northwest), which must directly follow it, specifies where we want the legend to by on our plot. The locations are given as cardinal directions: north, south, east, west, etc.

Note in the m-file in Example 2, the placement of the `LineWidth` and `MarkerSize` parameters inside the `plot()` function. Also, note the placement of the `title()`, `xlabel()`, `ylabel()`, and `legend()` functions, which are after the `plot()` function in the m-file.

How can I add a grid to my graph?

Using a grid in a figure is a helpful tool to make a graph more readable. However, using a grid is not always useful. In some cases, a grid can be counter-productive because the grid lines can crowd a plot. Placing a grid is very similar to using a title or axis label, just type `grid on`. Just like `hold off`, you can also use `grid off`.

Example 2

Plot the function $y = 2.1x + 4.4$ and the following data set on the same plot. Data pairs to plot are: (0, 4.12), (2, 8.6), (4, 11.5), (5, 15.3), (7, 18.0), (8.5, 21.25). Plot the function in the domain of 0 to 10 with an interval between the points of 0.1. Use blue points (markers) for the data set with a specified marker size of 6 and a red dotted line for the function with a line width of 2. Include a legend, title, and axis labels on the plot. Make the title in bold font and axis labels in italics.

Solution

Note: This is an extension of the solution given in Example 1.

MATLAB Code example2.m

```
clc
clear
close all

%----------------------------- PURPOSE -----------------------------
%To put multiple data sets on the same plot with professional formatting

%----------------------------- INPUTS ------------------------------
x = [0     2    4     5     7     8.5];    %Defining x points
y = [4.12  8.6  11.5  15.3  18.0  21.25];  %Defining y points

xFunc = 0:0.1:10;         %Generating the domain/independent variable
                          %    values for the function
yFunc = 2.1*xFunc + 4.4;  %Generating the range/dependent variable
                          %    values for the function

%------------------------ SOLUTION/OUTPUTS -------------------------
plot(x,y,'bo','MarkerSize',6)         %Plotting x and y points for data
                                      %    set vectors
hold on                               %Telling MATLAB to place new plots
                                      %    on the same figure
grid on                               %Putting grid on the plot
plot(xFunc,yFunc,'r:','LineWidth',2)  %Plotting x and y points for
                                      %    function vectors
hold off
```

MATLAB Code (continued) example2.m

```
%Adding plot formatting
legend('Data Points','Function: y = 2.1*x + 4.4',...
       'location','NW')
title('\bfy = 2.1*x + 4.4')    %Using bold font
xlabel('\itx Data Points')   %Using italics
ylabel('\ity Data Points')   %Using italics
```

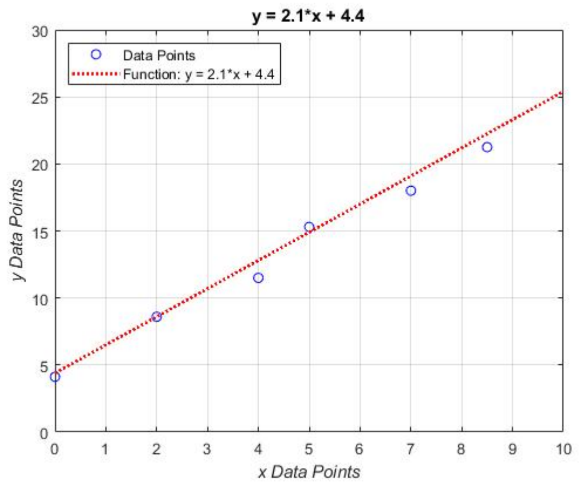


Figure 3: The figure output by the code in Example 2.

Similar to the title or axis label functions, the `grid` command needs to be placed after the `plot()` function in m-file. If `hold on` is used, the `grid` command should be directly after it as shown in Example 2.

How can I add special characters in my axis labels and title?

It may be necessary to add superscripts and subscripts to the item description(s) to the plot title, legend and/or labels. The syntax for superscripts and subscripts are placed where needed in the `title()`, `xlabel()`, `ylabel()`, and `legend()` functions. What is to be placed in the desired superscript or subscript must be inside braces `{}`, and the script character is placed before the braces. The character `_` is used for subscripts and the character `^` is used for superscripts. Similarly, one may use similar script statements as shown below for the axis labels, legends, etc. For example, to display, $y_1 = x^3$ in the title of a figure, one would type:

```
plot(x,y)
title('y_{1}= x^{3}')
```

The use of Greek letters or bold, italic and regular font may also be useful when making a graph. These can be added using a backslash followed by the desired font variation. To name a few, add bold font by using `\bf`, italic font by using `\it`, and regular font by using `\rm` followed by the desired text. To display Greek letters, one can use `\greek_letter` (see Table 4 and Example 3).

Table 4: Some of the special characters to be used with figures. The symbols can be used with any of the functions given as examples and more.

Symbol	Syntax	Example Usage
α	`\alpha`	`title('\alpha')`
ω	`\theta`	`xlabel('\theta')`
π	`\pi`	`ylabel('\pi')`
σ	`\sigma`	`legend('\sigma')`
τ	`\tau`	`xlabel('\tau')`
$\pm$	`\pm`	`ylabel('\pm')`
$\div$	`\div`	`legend('\div')`

For example, to place, $\mathbf{c} = \mathbf{2}^{*}\,{}^{*}\mathbf{r}$ in the title of a figure (notice the bold font), one would type `title('\bfc = 2*\pi*r')`. For a more complete list of special characters that can be used with your figures, conduct a MATLAB help search (keyword: Text Properties). Example 2 shows the use of a few special font styles, including bold font and subscripts in a plot.

How can I change axis limits and tick labels?

MATLAB makes it easy to adjust the limits of your axes to fit your data. The `xlim()` function adjusts the displayed domain of the horizontal axis, while the `ylim()` function adjusts the displayed range of the vertical axis.

For some data, such as the sinusoidal wave shown in Example 3, it can be useful to change the tick increments. `xticks()` and `yticks()` redefine the tick increment. That is, how far apart the ticks are on the axis. The corresponding tick labels can be changed with the `xticklabels()` and `yticklabels()` functions. These will allow you to change the tick label to any custom text compatible with MATLAB.

Example 3

Plot the function $f(t) = 0.3\sin(t)$ for time values of 0 to 2π in steps of 0.2. Set the x-axis ticks and tick labels to be from 0 to 2π in steps of $\pi/2$. Be sure to use the symbol (π) rather than just writing “pi”.

Solution

MATLAB Code example3.m

```
clc
clear
close all

%------------------------------ PURPOSE ------------------------------
%To demonstrate changing axis ticks and axis limits

%------------------------------ INPUTS ------------------------------
time    = 0:0.2:2*pi;      %Generating the domain/independent variable
                           %    values for the function
voltage = 0.3*sin(time);   %Generating the range/dependent variable
                           %    values for the function

%------------------------- SOLUTION/OUTPUTS -------------------------
figure                %Creating a blank figure
plot(time, voltage)   %Plotting the function across the specified domain

xlim([0 2*pi])                                         %Setting axis limits
xticks([0, pi/2, pi, 3*pi/2, 2*pi])                    %Setting the x tick
                                                       %   values
xticklabels({'0','\pi/2','\pi','3\pi/2','2\pi'})  %Setting x tick labels

%Note: the value \pi will result in the Greek symbol pi
```

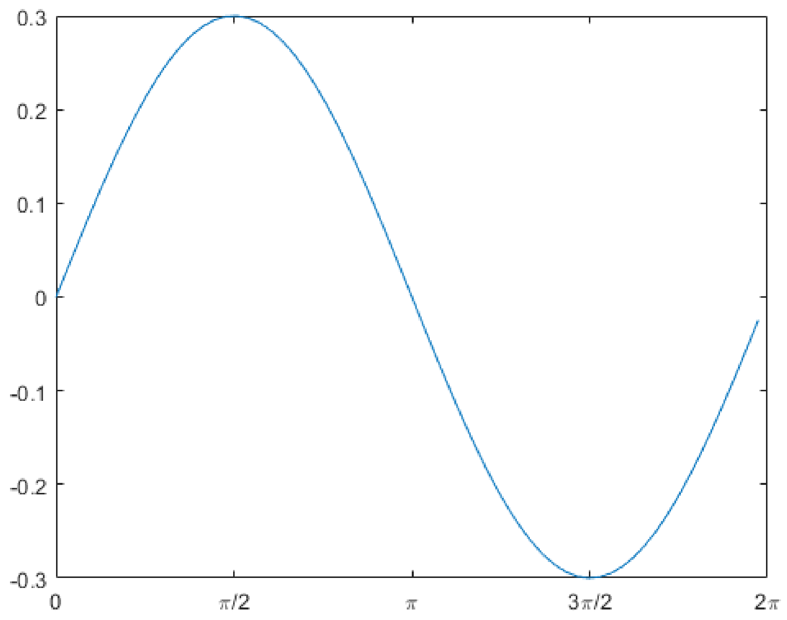


Figure 4: Plot with custom axis limits and axis labels on the horizontal axis (output for Example 3).

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Add a title to plot	`title()`	`title('Your title')`
Add x-axis label to plot	`xlabel()`	`xlabel('Your x label')`
Add y-axis label to plot	`ylabel()`	`ylabel('Your y label')`
Place a legend on the plot	`legend()`	`legend('plot1','plot2')`
Place a grid on the plot	`grid`	`grid on`
Set custom line width for a data set	`LineWidth`	`plot(x,y,'LineWidth',5)`
Set a custom marker size for a data set	`MarkerSize`	`plot(x,y,'MarkerSize',5)`
Set custom x limit for plot	`xlim()`	`xlim([lowerX,upperX])`
Set custom y limit for plot	`ylim()`	`ylim([lowerY,upperY])`
Set custom x ticks for plot	`xticks()`	`xticks([minTick,maxTick])`
Set custom y ticks for plot	`yticks()`	`yticks([minTick,maxTick])`
Set custom x tick labels for plot	`xticklabels()`	`xticklabels({'x1','x2'})`
Set custom y tick labels for plot	`yticklabels()`	`yticklabels({'y1','y2'})`

Multiple Choice Quiz

(1). The function to add a title to a plot is

(a) `ptitle()`

(b) `t()`

(c) `title()`

(d) `label()`

(2). The `MarkerSize` parameter

(a) adjusts the overall size of the figure font.

(b) adjusts the size of plotted points.

(c) changes the aspect ratio of the graph size.

(d) changes the thickness of plotted lines.

(3). To add a subscript, use the character(s)

(a) `n{}`

(b) `n()`

(c) `_{}`

(d) `_()`

(4). Which of the following will show the plot title in italics?

(a) `title('\it This is a plot title.')`

(b) `title('it{This is a plot title.}')`

(c) `title('it(This is a plot title.)')`

(d) `title('This is a plot title.\it')`

(5). Two sets of data points and a function, coded in the order, `data_set_1`, `data_set_2`, and `function_1`, are plotted. The correct code sequence to create the appropriate legend is

(a) `legend('data set 1','data set 2','function 1')`

(b) `legend('function 1','data set 2','data set 1')`

(c) `legend(data set 1, data set 2, function 1)`

(d) Code sequence does not matter.

Problem Set

(1). A rocket is horizontally strapped to the top of a sled and ignited. The position of this contraption is given as a function of time t (sec) by

$$s(t) = \frac{3}{50}t^3 + \frac{7}{30}t^2 - 5t \quad \text{(ft)}.$$

Plot the position of the sled in MATLAB from 0 to 60 seconds. Add a title (bold font) and axis labels (italic font) to the plot. Remember to follow the guidelines given in Lesson 3.1 for raising a vector to a power when plotting.

(2). Plot the lift and drag forces exerted on an airfoil as a function of velocity. Use velocity (v) values from 0 to 45 m/s on a log-linear plot (log-scale on the y-axis). The working fluid density (ρ) is 1.423 kg/m^3, the exposed airfoil area

(A) is 129 m^2, and the coefficients of drag (C_D) and lift (C_L) are 0.178 and 0.896, respectively. Recall that the equations for drag and lift forces are

$$F_D = \frac{1}{2} C_D \; A \; \rho \; V^2,$$

$$F_L = \frac{1}{2} C_L \; A \; \rho \; V^2,$$

Your plot should display an appropriate legend, title, axis labels, and units. The line width of the two lines should be adequately sized. Remember to follow the guidelines given in Lesson 3.1 for raising a vector to a power when plotting.

(3). The required specific input work (kJ/kg) for an insulated refrigerant compressor is found to be,

$$w_{in} = h_{out} - h_{in}$$

where h_{out} and h_{in} have units of kJ/kg and correspond to the enthalpies at the exit and inlet of the compressor, respectively. The inlet enthalpy is given as a constant value of 278.76 kJ/kg. On the other hand, exit enthalpy will change as a function of exit pressure and temperature. The following data is collected:

$$\text{Exit pressure (bar)} = [1.0 \;\; 1.4 \;\; 1.8 \;\; 2.0 \;\; 2.4 \;\; 2.8 \;\; 3.2]$$

$$h_{out}\left(\frac{kJ}{kg}\right) = [278.76 \;\; 286.96 \;\; 295.45 \;\; 304.50 \;\; 313.49 \;\; 332.60 \;\; 342.21]$$

Plot the input work as a function of the exit pressure, and show the data as points (use circles) on a standard linear plot. Add a figure title and axis labels, use increased marker size, and show a grid.

(4). Torque, T, is given by $T = F \cdot r$ where F is the force and r is the radius (moment arm). Create two plots on two separate figures. In one figure, plot the torque vs. radius. Use a constant value of $F = 5$ N and radius values of 0 to 10 m with a step size of 0.2. In the second figure, plot the force vs. radius.

Let the torque be a constant value of 5 Nm and the same radius values as the first plot. You can rearrange the torque equation to solve for force: $F = \dfrac{T}{r}$. Include a title, axis labels, and a grid on both figures. Remember to follow the guidelines given in Lesson 3.1 for a vector in the denominator when plotting.

(5). Given the data set below for stress and strain for a uniaxial text on a unidirectional composite material, create a 2D line plot of stress vs. strain. Use a solid, green line with an appropriate line width. Also include a grid, axis labels, axis limits, and units to improve the appearance and effectiveness of your plot.

Table A: Stress vs. strain for a composite material.

Stress (MPa)	**Strain (%)**
0	0
306	0.183
612	0.36
917	0.5324
1223	0.702
1529	0.867
1835	1.0244
2140	1.1774
2446	1.329
2752	1.479
2767	1.5
2896	1.56

Module 3: PLOTTING

Lesson 3.3 – Advanced Plotting

Learning Objectives

1) *represent data in a bar graph,*
2) *create line and surface plots in three dimensions,*
3) *graph data in a polar plot.*

Does MATLAB have more plotting capabilities?

In the previous lessons in this module (Lessons 3.1 and 3.2), we learned how to create and work with plots represented in two dimensions. However, sometimes you may want to show the data in a different type of plot. Although we will only cover a few of the more common advanced plotting capabilities in this lesson, MATLAB offers many other useful plotting formats. In this lesson, we cover bar graphs, 3D line and surface plots, and polar plots. This will give you a very good foundation to branch out to other options MATLAB offers if you need to do so in the future.

How can I create a bar graph?

Bar graphs are a great means to show quantities across groups. For example, a bar graph may be used to display how many students have received a certain discrete grade on an exam.

MATLAB uses the `bar()` function to create a bar graph. We demonstrate how to create a bar graph in Example 1 as well as change some essential parameters. In the first graph, there is a one-to-one correlation between groups and their values. However, in the second graph, we have a row of y data for each x data point. This results in "bar series", which you can see in the graph output (Figure 1).

You can review the documentation for the syntax specifics on a number of functional and formatting options like line color (outlines, fills, etc.), width, and style. Try to play around with the code and change the values of each of the inputs to get a sense of what aspect of the final output bar graph changes! One important additional thing to know is that `bar()` does not accept strings as categories directly. This means if your categories are strings (Example 1 has numbers as categories), you will need to use the function 'categorical()' to convert your categories (strings) into an appropriate data type. You can review the MATLAB documentation for `bar()` to see an example of this case.

Example 1

Display the data sets given below (Data set 1 and Data set 2) with two separate bar graphs. The "groups" (horizontal axis) are "21", "22", "23". Make all the bars black for data set 1, and use three different bar colors for data set 2. Put a title and axis labels on the figure for data set 2.

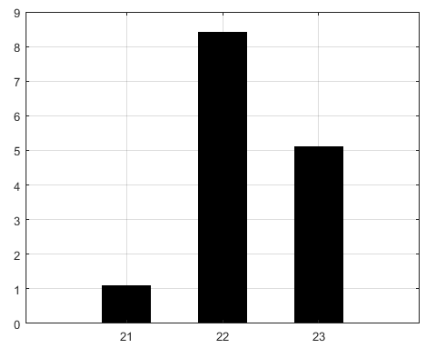
Data Set 1: $\begin{bmatrix} 1.1 & 8.4 & 5.1 \end{bmatrix}$

Data Set 2: $\begin{bmatrix} 1.1 & 1.4 & 1.6 \\ 8.1 & 8.8 & 8.4 \\ 5.5 & 5.7 & 5.6 \end{bmatrix}$

Solution

MATLAB Code example1.m

```
clc
clear
close all

%----------------------------- PURPOSE -----------------------------
%To visualize data on bar graphs

%----------------------------- INPUTS ------------------------------
xGroups  = [21 22 23];     %Groups/categories (independent position)
yValues1 = [1.1 8.4 5.1];  %Group values for first bar graph

yValues2 = [1.1 1.4 1.6;   %Group values for second bar graph
            8.1 8.8 8.4;
            5.5 5.7 5.6];

%------------------------ SOLUTION/OUTPUTS -------------------------
figure                                       %Creating a blank figure to
                                             %    plot on
bar(xGroups, yValues1, 0.5, 'grouped', 'k')  %Making the first bar graph
grid on                                      %Display a grid on the plot

figure                                  %Creating a second blank figure to
                                        %    plot on
bar(xGroups, yValues2, 0.9, 'grouped')  %Making the second bar graph
                                        %    (Note, color not specified)
grid on                                 %Display a grid on the plot

%title and labels work the same here as in 2D line plots (appear on 2nd
%    figure only
title('Example of a bar graph')
xlabel('Bar Graph Labels')
ylabel('Bar Graph Values')
```



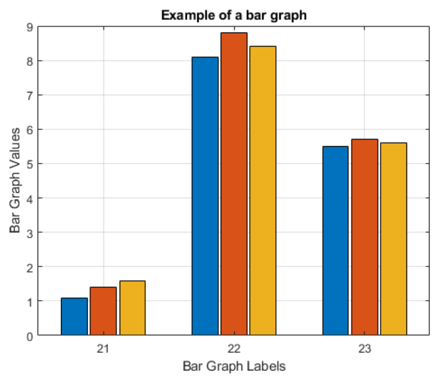


Figure 1: Figure outputs for Example 1.

How can I create a 3D line plot?

Plotting in three dimensions can be useful when visualizing a function with two independent variables which could be over space, time, or any other changing quantity (e.g., $z(x, y)$, $y(x, t)$, etc.). For example, it may be useful to visualize the speed of a car over time and across a number of starting speeds (using $x(t, v_i) = v_i - a \cdot t$). To do this, MATLAB makes plotting in three dimensions very straightforward with both 3D line plots and 3D surface plots.

As the name suggests, a 3D *line* plot shows a line drawn across three dimensions. A 3D line plot is generated with the `plot3()` function as seen in Example 2. Note that the `plot3()` function is just like the conventional 2D `plot()` function from previous lessons; however, it needs one additional dependent variable parameter in the third dimension.

Sometimes, you may wish to set the aspect ratio between axes to be 1:1:1. You can do this using the `daspect()` function, which allows you to adjust the ratio between all three axes.

Example 2

Create a 3D line plot from the functions $x = t$, $y = 2\sin(4t)$, and $z = \cos(4t)$ for t values of 0 to 5 with an interval of 0.01. Note that any axis can be independent/dependent on a 3D plot.

Solution

MATLAB Code example2.m

```
clc
clear
close all

%----------------------------- PURPOSE -----------------------------
%To create a 3D line plot
```

MATLAB Code (continued) example2.m

```
%------------------------------ INPUTS ------------------------------
t = 0:0.01:5;     %Time vector from 0s to 5s at 0.01s interval.
                  %    Note that this is sometimes referred to as the
                  %    parametrization variable
X = t;            %Defining the domain of the data (independent variable)
Y = 2*sin(4*t);   %Defining the dependent variable, Y
Z = cos(4*t);     %Defining the dependent variable, Z

%------------------------- SOLUTION/OUTPUTS -------------------------
figure                %Creating the blank figure for plot
plot3(X, Y, Z, 'k')   %Creating the 3D Line plot

grid on               %Show the plot grid
xlabel('X Axis')      %Label the x-axis
ylabel('Y Axis')      %Label the y-axis
zlabel('Z Axis')      %Label the z-axis
daspect([1 1 1])      %Set the aspect ratio to 1 for each axis with the
                      %    daspect() function. This ensures that the plot is
                      %    not scaled or skewed.
```

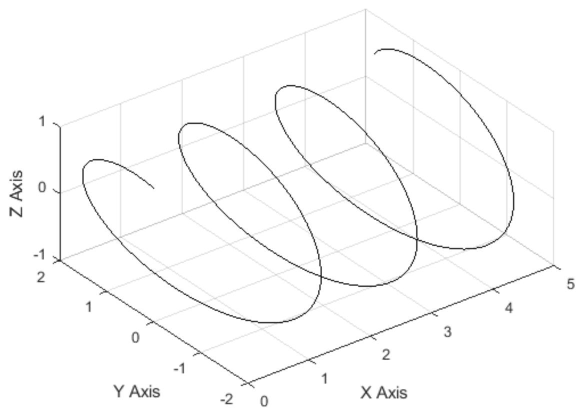


Figure 2: Figure output of a 3D line plot for Example 2.

How can I create a 3D surface plot?

Visualizing a *surface* in 3D requires that all surface points that we want to plot to be defined in 3D (of course). This is not as simple as giving three vectors like we did for plotting a 3D line because we now have a surface to plot. You could think of this as many lines stacked against each other. There are two main steps to getting the coordinates we need for a 3D surface. The first part of this process is to create a grid of independent coordinate points. Next, we use these 2D points to generate the third dimension, which is similar to generating a vector for a 2D function like we saw in previous lessons in this module (Lessons 3.1 and 3.2).

Fortunately, MATLAB provides the `meshgrid()` function to automate the process of creating a mesh of surface points. We only need to give a vector input for each independent variable (dimension), and `meshgrid()` will do the rest! Figure 3 shows an example of how `meshgrid()` creates coordinate grids (matrices) from vector inputs.

Once the two coordinate grids (X and Y) are obtained, those two matrices are used to create a third matrix of points as shown in Figure 3. This third matrix will be plotted as the dependent variable (i.e., *Z(X,Y)*) on the 3D plot. You can see further explanation and documentation of this process for `meshgrid()` in MATLAB documentation.

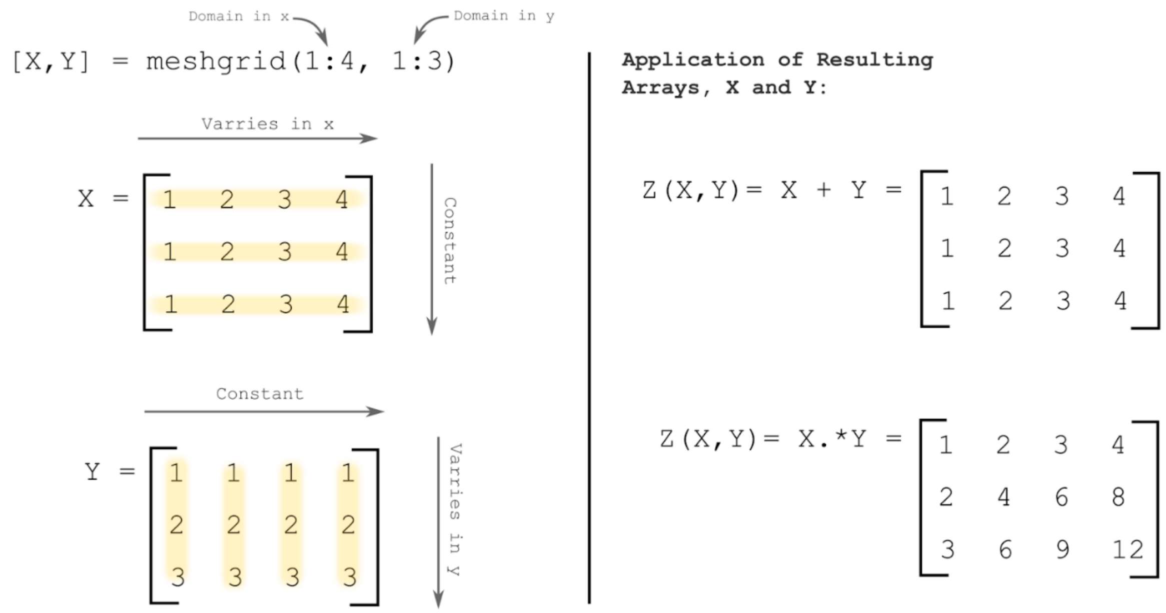

Domain in x
Domain in y
[X,Y] = meshgrid(1:4, 1:3)
Varries in x
X = [1 2 3 4; 1 2 3 4; 1 2 3 4]
Constant
Constant
Y = [1 1 1 1; 2 2 2 2; 3 3 3 3]
Varries in y
Application of Resulting Arrays, X and Y:
Z(X,Y)= X + Y = [1 2 3 4; 1 2 3 4; 1 2 3 4]
Z(X,Y)= X.*Y = [1 2 3 4; 2 4 6 8; 3 6 9 12]

Figure 3: Visual demonstration of how `meshgrid()` generates coordinate grids, X and Y, (matrices) from range vector inputs. The right side shows the application of these grids in order to generate a dependent function *Z(X,Y)* in three dimensions.

After each of the three coordinate grid matrices is created (two independent and one dependent), the `surf()` function can be used to plot the surface coordinates (surface defined by the function) as shown in Example 3. Note that you can change the color theme of the surface with the `colormap` command. The different color themes that can be used with `colormap` are given in Table 1. There are many more color theme options than we can show here including making complete custom theme colors.

Table 1: Some common color themes for the `colormap` command to use with 3D surface plots.

Color Theme & Syntax	Example Usage
`spring`	`colormap spring`
`summer`	`colormap summer`
`autumn`	`colormap autumn`
`winter`	`colormap winter`
`hot`	`colormap hot`
`gray`	`colormap gray`

Example 3

Create a 3D surface plot of the function $Z = \cos(X) + 0.1Y + 1$. Use values from -4 to 8 for the X variable with a step size of 0.3. For variable Y, use values from 0 to 10 with a step size of 0.5. Specify the `colormap` theme 'autumn'.

Solution

At the end of the example code, we use the function `daspect()`. Using `daspect()` to manually set the aspect ratio for axes ensures that the figure will not be skewed or stretched in a way that we do not want.

MATLAB Code example3.m

```
clc
clear
close all

%------------------------------ PURPOSE -----------------------------
%To create a 3D surface plot

%------------------------------ INPUTS ------------------------------
xPoints = -4:0.3:8;  %Defining the x domain of the data (indep. variable)
yPoints = 0:0.5:10;  %Defining the y domain of the data (indep. variable)

%Producing a 2D array to represent all possible combinations across the
%    x and y domains.
[X, Y] = meshgrid(xPoints, yPoints);

%Creating a matrix from the Z(X, Y) function
Z  = cos(X) + 0.1*Y + 1;

%------------------------- SOLUTION/OUTPUTS -------------------------
figure            %Creating a blank figure
surf(X,Y,Z)       %Plotting the Z(X, Y) surface

colormap autumn   %Set 'colormap' of surface to 'autumn' theme
grid on           %Show the plot grid
xlabel('X Axis')  %Label the x-axis
ylabel('Y Axis')  %Label the y-axis
zlabel('Z Axis')  %Label the z-axis
daspect([1 1 1])  %Set the aspect ratio to 1 for each axis.
```

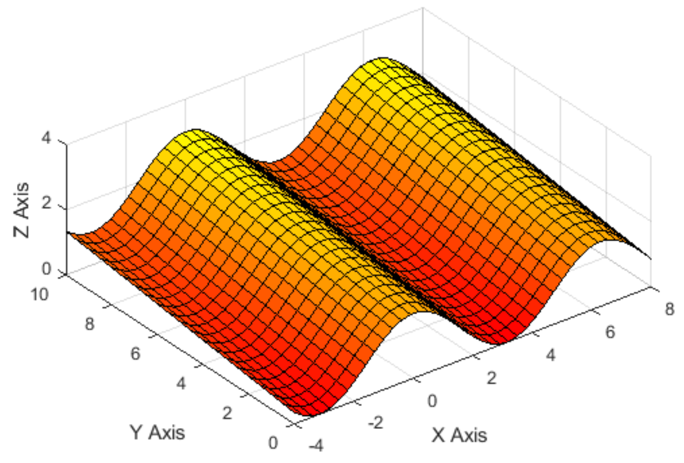


Figure 4: Figure output of a 3D surface plot for Example 3.

Similar results to Figure 4 of creating a 3D surface can be obtained by using the `mesh()` or `contour()` function instead of the `surf()` function. The `mesh()` function (not to be confused with `meshgrid()`) creates a wire surface, while the `contour()` function creates a 2D contour map (similar to a heat map) of the surface.

How can I create a polar plot?

Some data will require you to visually analyze functions in the polar (or radial) coordinate system, where data points vary around some center point. For example, the forces around a stationary wheel vary around the wheel (i.e., force is a function of angle).

We can plot such 2D polar/radial functions with the `polarplot()` function (use `polar()` for previous versions (before R2016a)), which accepts an angle vector (independent variable in Example 1) and a radius vector (dependent variable in Example 1). Line properties such as color or width can be changed in the same way used in `plot()`.

✓ ***Important Note:*** The angle vector for `polarplot()` must be in radians: not degrees.

Example 4

Plot the function $r = 2\cos(5\theta)$ on a polar plot. Use theta values from 0 to $3\pi/2$ with an interval of 0.01. Set appropriate limits on the plot.

Solution

MATLAB Code example4.m

```
clc
clear
close all

%------------------------------ PURPOSE ------------------------------
%To create a 2D polar plot

%------------------------------ INPUTS ------------------------------
theta  = [0 : 0.01 : 3*pi/2];  %Defining the indep. variable vector (radians)
radius = 2*cos(5*theta);       %Defining the dep. variable vector (radius)

%------------------------- SOLUTION/OUTPUTS -------------------------
figure                                          %Creating a blank figure
polarplot(theta, radius, 'k', 'LineWidth', 2);  %Creating the polar plot
rlim([0 2.5]);                                  %Setting polar plot limits
```

While polar plots function similar to “regular” 2D Cartesian plots, changing specifics of the plot like the grid, labels, font, spacings, etc. require a different set of axis properties called polar axis properties. For example, `ylim()` is `rlim()` and `yticks()` is `rticks()` for polar axes. You can see an application for this in Example 4.

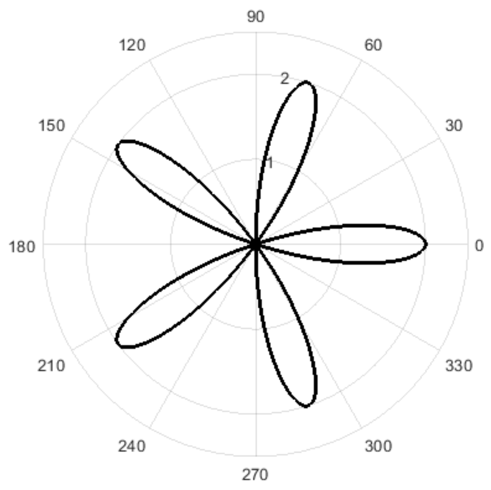


Figure 5: Figure output of a 2D polar plot for Example 4.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Create a bar graph	`bar()`	`bar(groups,values)`
Format group names to be used with `bar()`	`categorical()`	`categorical({'group1', 'group2'})`
Create a 3D line plot	`plot3()`	`plot3(x,y,z)`
Set aspect ratio between plot axes	`daspect()`	`daspectdaspect([1,1,1])`
Creates mesh of surface points for 3D surface plot	`meshgrid()`	`mesh grid([1:10],[1:10])`
Plot surface coordinates	`surf()`	`surf(x,y,z)`
Give surface a color theme	`colormap`	`colormap winter`
Create a polar plot	`polarplot()`	`polarplot(theta,r)`
Set custom limits on the radial axis	`rlim()`	`rlimrlim([rMin, rMax])`
Set custom limits on the angular axis	`thetalim()`	`thetalim([thetaMin, thetaMax])`
Set custom tick values for the radial axis	`rticks()`	`rticks({'r1','r2','r3'})`
Set custom tick values for the angular axis	`thetaticks()`	`thetaticks({'t1','t2'})`

Multiple Choice Quiz

(1). To create a 3D line plot, the correct function and inputs are

(a) `plot3(x,y,z)`

(b) `plot3d(x,y,z)`

(c) `plot(x,y,z)`

(d) `3D(x,y,z)`

(2). To create a 3D surface plot, the correct function is

(a) `surf()`

(b) `contour()`

(c) `surfplot()`

(d) `surf3d()`

(3). For a 3D surface plot, the inputs should be

(a) matrices

(b) vectors

(c) integers

(d) strings

(4). To place a label on the x-axis of a 3D plot, one should use

(a) `xlabel()`

(b) `xlabel3()`

(c) `label1()`

(d) `label3d()`

(5). For a 3D line plot, the inputs should be

(a) matrices

(b) vectors

(c) integers

(d) strings

Problem Set

(1). Create a bar graph of the movies that are currently in the top five for the highest domestic, opening weekend box office numbers (given below in Table A). Make the bars on the graph red and be sure to make your graph look nice. Add a title, change the y-axis tick labels to be in millions (e.g., $200 million would be "200"), and y-axis label (be sure to note the scale is in millions).

Table A: Top five box office numbers (domestic) for the opening weekend of the movie.

Movie Title (abbreviated)	Avengers: Infinity War	The Force Awakens	The Last Jedi	Jurassic World	The Avengers
Opening Weekend Box Office ($)	257,698,183	247,966,675	220,009,584	208,806,270	207,438,708

(2). Plot the parametric functions $x = 2t$, $y = \cos(3t)$, $z = \sin(2t)$ using values for t of 0 to 10 with an interval of 0.1. Use a red line with a line width of 3.

Try exploring the plot by clicking Tools>Rotate 3D from the figure window menu. Note how the plot looks if you rotate it to view only the XY or XZ planes.

(3). Plot the hyperbolic function$f(x, y) = x^2 + y^2$ using values from -3 to 3 with an interval of 0.1 for x and y. Choose a color theme that you like and put a grid on the plot. Place axis labels on all three axes. Remember to use the array operator for the exponents in this problem (Lesson 3.1).

(4). Plot the equation $z = \ln(x^2 + y^2)$ using values from -2 to 2 with an interval of 0.1 for x and y. Choose a color theme that you like and put a grid on the plot. Place axis labels on all three axes. Remember to use the array operator for the exponents in this problem (Lesson 3.1).

(5). Plot the function for theta values of 0 to 4 in steps of 0.01 using a polar plot. Change angular (theta) axis ticks and tick labels to be aligned with 0, $\pi/4$, $\pi/2$, $3\pi/4$, etc. Be sure to use the symbol for pi.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.1: Basic Algebra, Logarithms, and Trigonometry

Learning Objectives

After reading this lesson, you should be able to:

(1) *perform basic algebra with mathematical operators,*

(2) *find logarithmic functions with different bases in MATLAB,*

(3) *calculate basic trigonometric functions in MATLAB.*

What kind of mathematical functions and operations are available in MATLAB?

Basic math (addition, subtraction, multiplication, etc.) syntax in MATLAB is the same as it is in most calculators (`+,-,*,/,^`). The syntax is the same whether used with variables or directly with numbers. You have seen these numerous times in previous lessons, but we include them here for completeness.

Important Note: You must use the multiplication operator *everywhere* you have multiplication. $7x$ should be `7*x`, and $7(x+1)$ should be `7*(x+1)`. You will get an error if you are missing a multiplication operator.

MATLAB has several mathematical functions such as logarithmic, exponential, trigonometric, hyperbolic, etc. In this lesson, we will cover the use of logarithmic and trigonometric functions.

How do I use logarithmic functions in MATLAB?

The first function that is presented is the `log()` function. This function is only for the natural logarithm.

The inverse of a natural log function is the exponential function. You <u>cannot</u> use `log^-1` in MATLAB to find the inverse of a natural log. The function to find the value of an exponential function is `exp()`.

Example 1

Find the natural log of 4 and 0.2, and name these outputs $y1$ and $y2$, respectively. Find the value of the exponential function of both $y1$ and $y2$, and examine the results.

Solution

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To evaluate the natural log and value of the exponential function at given
%    numbers
fprintf('To evaluate the natural log and value of the exponential\n')
fprintf('function at given numbers.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining some numbers to work with
a = 4;
b = 0.2;
fprintf('The inputs are the arbitrary numbers %g and %g.\n\n',a,b)

%------------------------------ SOLUTION ------------------------------
%Evaluating the natural log for both numbers
ln1 = log(a);
ln2 = log(b);

%Evaluating the exponential function for both numbers
exp1 = exp(a);
exp2 = exp(b);

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('Evaluating the expressions gives us:\n')
fprintf('ln(%g) = %g and ln(%g) = %.5f.\n\n',a,ln1,b,ln2)

fprintf('Evaluating the expressions gives us:\n')
fprintf('e^%g = %g and e^%g = %.5f.\n',a,exp1,b,exp2)
```

```
Command Window Output                                         Exam
```

```
PURPOSE
To evaluate the natural log and value of the exponential
function at given numbers.

INPUTS
The inputs are the arbitrary numbers 4 and 0.2.

OUTPUTS
Evaluating the expressions gives us:
ln(4) = 1.38629 and ln(0.2) = -1.60944.

Evaluating the expressions gives us:
e^4 = 54.5982 and e^0.2 = 1.22140.
```

Note that when typing the function exp(num), it cannot be substituted by exp^num, as this will give you a syntax error as given below.

```
??? Error using ==> exp

Not enough input arguments.
```

What about a logarithm that is not natural?

Besides the natural logarithm, one of the most common logarithms is log with the base 10. The function to find a log with a base of 10 is `log10()`.

To find the value of a logarithm that does not have a base of 10 or e (natural log), you may use the change of base formula. To simplify the m-file when using the change of base formula, it is recommended that you use natural logs to change the base. The formula for the log of b with the base a is,

$$\log_a(b) = \frac{\ln(b)}{\ln(a)}$$

Example 2

Using MATLAB, find:

(a) $\log_{10}(3)$

(b) $\log_8(3)$

Use the fprintf() function to describe the outputs in the Command Window. Hint: Consider using the change of base formula for part (b).

Solution

Note the different code used for finding the natural log (Example 1) and the log with a different specified base (Example 2). The functions for natural log, exponential function, and $\log_{10}$ can all be found in Table 1 at the end of this lesson.

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To logarithms of different bases of given numbers
fprintf('To logarithms of different bases of given numbers.\n\n')
```

MATLAB Code (continued) example2.m

```
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
a = 3;        %Defining a number to work with
base1 = 10;  %Defining the first base
base2 = 8;   %Defining the second base
fprintf('The desired bases are %g and %g.\n',base1,base2)
fprintf('The desired argument of the logarithms is %g.\n\n',a)

%------------------------------ SOLUTION ------------------------------
log1 = log10(a);            %Evaluating the log base 10
log2 = log(a)/log(base2);   %Evaluating the log base 8

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('Evaluating the expressions gives us:\n')
fprintf('a)    log_%.0f(%g)  = %.4f\n',base1,a,log1)
fprintf('b)    log_%.0f(%g)   = %.4f\n\n',base2,a,log2)
```

```
Command Window Output                                        Exam
PURPOSE
To logarithms of different bases of given numbers.

INPUTS
The desired bases are 10 and 8.
The desired argument of the logarithms is 3.

OUTPUTS
Evaluating the expressions gives us:
a)     log_10(3) = 0.4771
b)     log_8(3)  = 0.5283
```

How can MATLAB evaluate trigonometric functions?

MATLAB has several trigonometric functions. However, note that the input for these trigonometric functions is in radians: not degrees.

The function to find the cosine of an angle is `cos()`. This function assumes an input of an angle in radians. The functions for sine, tangent, cosecant, secant, and cotangent are `sin()`, `tan()`, `csc()`, `sec()`, and `cot()`, respectively. These functions have the same inputs and usage as the `cos()` function shown above (see Table 1).

If you wish to input the argument to a trigonometric function in degrees, attach a d to the end of the function. For example, to find the value of the cosine of 30 degrees, you may type, `cosd(30)`.

Example 3

A right-angled triangle ABC, shown in Figure 3, with $x = 50^{\circ}$ and $\overline{\text{AB}} = 3$ is given.

(a) Convert x to radians.

(b) Find $\overline{\text{BC}}$.

(c) Find $\overline{\text{AC}}$.

(d) What is the sum of ∠BAC and ∠ABC in degrees?

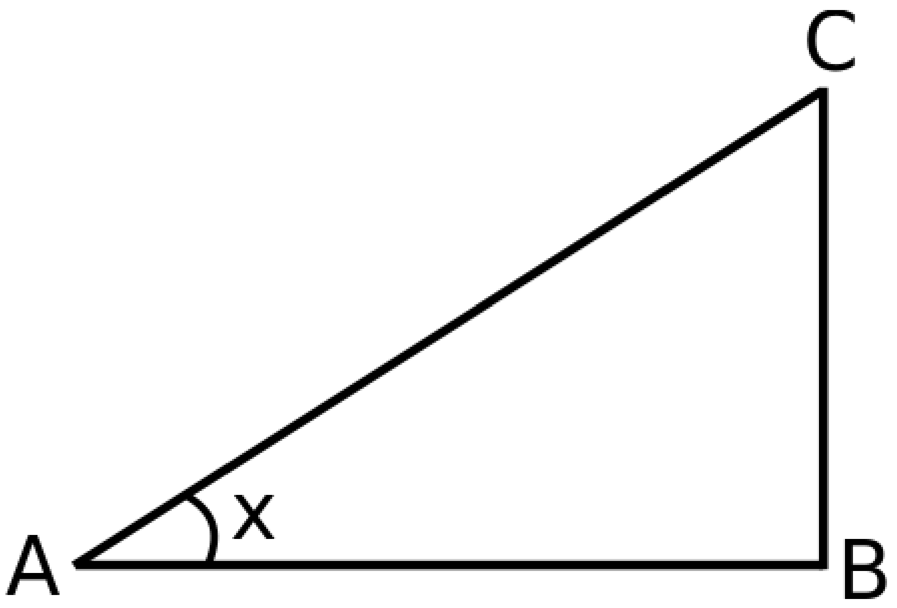


Figure 1: The Command Window output for Example 3.

All outputs should be displayed in the Command Window using the `fprintf()` or `disp()` functions.

Solution

Notice, in the example code, that MATLAB stores the numeric value of π in the predefined variable `pi`. Also take a look at the method to convert the value of x from degrees to radians for part a (if you are not familiar with it). Be sure that the argument for all applicable trigonometric functions is in radians or add a d for degrees (e.g., `sind(30)`) as mentioned previously.

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To calculate trig expressions
fprintf('To calculate trig expressions.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
x  = 50;  %Arbitrary angle in degrees
AB = 3;   %A measured length

fprintf('The inputs are:\n')
fprintf(' x = %g deg \n AB = %g \n\n',x,AB)
```

MATLAB Code (continued) examp

```
%------------------------------ SOLUTION ------------------------------
% a)  convert value of x to radians
xr = x*pi/180;

% b)  find BC
BC = AB*tan(xr);

% c)  find AC
AC = AB/cosd(x);

% d)  sum of the angles BAC and ABC in degrees
sumAng = x + 90;

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('a) The angle %g deg is %g rads.\n',x,xr)
fprintf('b) The length of BC is %.4f.\n',BC)
fprintf('c) The length of AC is %.4f.\n',AC)
fprintf('d) Sum of the angles is %.3f deg.\n',sumAng)
```

```
Command Window Output                                  Example 3
PURPOSE
To calculate trig expressions.

INPUTS
The inputs are:
 x = 50 deg
 AB = 3

OUTPUTS
a) The angle 50 deg is 0.872665 rads.
b) The length of BC is 3.5753.
c) The length of AC is 4.6672.
d) Sum of the angles is 140.000 deg.
```

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Find the natural log of a number	`log()`	`log(a)`
Evaluate the exponential function at a number	`exp()`	`exp(a)`
Log with base 10	`log10()`	`log10(a)`
Sine of an angle	`sin()`	`sin(a)`
Cosine of an angle	`cos()`	`cos(a)`
Tangent of an angle	`tan()`	`tan(a)`
Cosecant of an angle	`csc()`	`csc(a)`
Secant of an angle	`sec()`	`sec(a)`
Cotangent of an angle	`cot()`	`cot(a)`
Sine inverse of a number	`asin()`	`asin(a)`
Cosine inverse of a number	`acos()`	`acos(a)`
Tangent inverse of a number	`atan()`	`atan(a)`
4 quadrant tan inverse complex number	`atan2()`	`atan2(Im,Re)`

Multiple Choice Quiz

(1). The function for finding the natural log of a number is

(a) `ln()`

(b) `log()`

(c) `loge()`

(d) `nlog()`

(2). The function for finding the exponential function of a number is

(a) `e()`

(b) `e\^()`

(c) `exp()`

(d) `exp\^()`

(3). The function to find the value of $\sin(a)$ where a is 34°, is

(a) `sin(34)`

(b) `sine(34)`

(c) `sin((34\*180)/pi)`

(d) `sin((34\*pi)/180)`

(4). The function `acos()`

(a) determines if the cosine value can be evaluated.

(b) evaluates the cosine of an angle in degrees.

(c) evaluates the cosine of an angle in radians.

(d) evaluates the inverse cosine of an argument.

(5). The `sind()` function

(a) determines if the sine value can be evaluated.

(b) evaluates the inverse sine of an argument.

(c) evaluates the sine of an angle given in degrees.

(d) evaluates the sine of an angle given in radians.

Problem Set

(1). Given that $a = 7$, $b = 2$, and $c = 11$, using MATLAB, find the values of

(a) $\log_{10}(b)$

(b) $b\ln(c)$

(c) $e^{-\frac{a}{2}}$

(d) $\log_2(a)$

Output each solution to the Command Window, and check your results using a calculator.

(2). The approximate value of e^x (exponential function) can be found by using a finite number of terms of the following infinite Maclaurin series,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

for all x.

Complete the following using $x = 2.3$:

(a) Compare the output of the first 3 terms for the Maclaurin series for exponential function against the MATLAB output for the exponential function.

(b) Redo part (a) using first 6 terms of the Maclaurin series.

Make sure to use the `fprintf()` or `disp()` functions to display your program outputs in the Command Window.

(3). Given the triangle ABC, shown in Figure A, with angles $\theta_1 = \frac{\pi}{8}$ radians, $\theta_2 = 34^\circ$, and length $\overline{\text{AB}} = 11$, find the following.

(a) θ_3 in degrees

(b) $\overline{\text{BC}}$

(c) $\overline{\text{AC}}$

(d) area of triangle ABC

Hint: The law of cosines is given as

$$\frac{\overline{\text{AB}}}{\sin(\theta_1)} = \frac{\overline{\text{BC}}}{\sin(\theta_2)} = \frac{\overline{\text{AC}}}{\sin(\theta_3)}$$

The area of a triangle is

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where s = semi-perimeter of the triangle and a, b, c = length of the three sides of the triangle

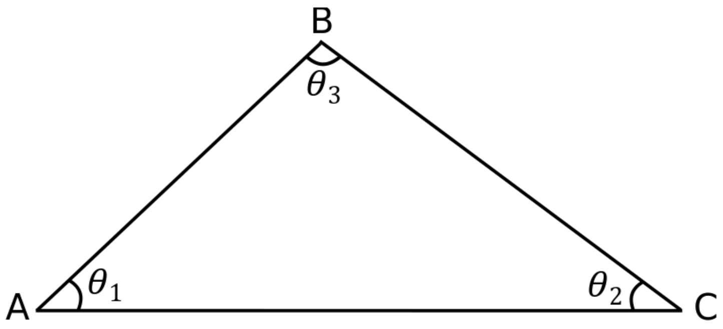


Figure A: Labeled triangle ABC is shown (not to scale).

Output each solution to the Command Window and verify your results > with a calculator. Make sure to use the `fprintf()` or `disp()` functions > to display your program outputs in the Command Window.

(4). The approximate value of $\sin(x)$ can be calculated using a finite number of terms of the following infinite Maclaurin series

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

for all x. Complete the following using 38°

(a) Compare the output of the first 3 terms of the Maclaurin series provided, against the MATLAB output for the `sine` command.

(b) Redo part (a) using the first 5 terms of the series.

Make sure to use the `fprintf()` or `disp`()‘ functions to display your program outputs in the Command Window.

(5). The mechanical components of a certain suspension system dynamically respond to an applied force by vibration. The actual position, $x(t)$, of the center of mass of the system as a function of time, t, is given by,

$$x(t) = Xe^{\xi\omega_\mathrm{n}t}(\cos(\sqrt{1-\xi^2}\omega_\mathrm{n}t))$$

Ideally, the center of mass would follow the following position function,

$$x(t) = Xe^{-\xi\omega_{\mathrm{n}}t}$$

Given that $\quad X = 2, \quad f_{\mathrm{n}} = 1.3 \quad \omega_{\mathrm{n}} = 2\pi f_{\mathrm{n}} \quad \xi = 0.1$

Plot the actual and the ideal position of the system as a function of time. Use a legend, give axis and graph titles, and use reasonable line thicknesses and symbols. Plot the position for the time from 0 to 8 seconds in a single plot where the horizontal axis is the time and the vertical axis is the position of the center of mass.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.2: Symbolic Variables

Learning Objectives

After reading this lesson, you should be able to:

1) *use symbolic variables in MATLAB to form equations,*
2) *substitute a number or variable into a symbolic expression,*
3) *convert symbolic variables to other data types.*

What is a symbolic variable?

A symbolic variable does not have a numerical value assigned to it. Use the following function as an example $f(x) = x^2$; where the symbolic variable is x. The value of x has no numeric value assigned to it; therefore, it is a symbolic variable.

The symbolic toolbox introduces syms: a new data type. syms allows you to create variables without assigning them a value (number, string, etc.). This is very useful when doing anything beyond the basic “2+2” math. Solving equations, calculus, and differential equations are a few examples of how syms is used to solve mathematical problems, and it demonstrates part of the functionality of the MATLAB symbolic toolbox.

What is a MATLAB toolbox?

A MATLAB toolbox, more traditionally referred to as a package or library in other programming languages, is just a set of code (which you do not need to see) that gives you more predefined functions. We have been using functions all along, but have not talked explicitly about a "toolbox" because we have used pre-installed, default functions. One example is the `plot()` function. MATLAB knows how to plot the data you input to the function because there is an algorithm doing the work for you behind the scenes. We will learn how to write our own functions and reference them in Module 7.

How do I use symbolic variables?

To use a symbolic variable in MATLAB, you use the syms command. This command needs to be placed before the symbolic variable that is intended to be used in an expression. Otherwise, MATLAB will return an "undefined variable" error. The syms command tells MATLAB to treat a specified variable as symbolic (it has no value). Note that symbolic variables must follow the same naming rules as all other variables (reference Lesson 2.1 for naming rules).

If more than one symbolic variable is required, the variables can be made symbolic by using the syms command by separating each symbolic variable by a space. Look at Example 1 to see the syms command in action.

Example 1

Given two functions $y(t)$ and $z(t,x)$, $y = t^2 + 2t$ and $z = \dfrac{tx+1}{x^2}$, use the syms command and display each function in the Command Window.

Solution

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To create symbolic variables and mathematical functions
fprintf('To create symbolic variables and mathematical functions.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
syms t x                %Creating symbolic variables

disp('Example function 1')
y = t^2 + 2*t           %Defining the first given mathematical function

disp('Example function 2')
z = (t*x + 1)/x^2       %Defining the second given mathematical function
```

Command Window Output Example 1

```
PURPOSE
To create symbolic variables and mathematical functions.

INPUTS
Example function 1

y =

t^2 + 2*t

Example function 2

z =

(t*x + 1)/x^2
```

Example 1 shows the syms command being used to make two variables, t and x symbolic. These symbolic variables are then used to generate the mathematical functions y and z. The Command Window output shows the two specified functions being output to the Command Window (they were not suppressed). Another way to direct output mathematical functions containing symbolic variables is to use the use the `disp()` function (e.g., `disp(y)`).

How do I clear specific variables?

To make a variable no longer symbolic, the clear command is used. You may recall that the first two lines of most m-files should be clc and clear. We know that using clear will clear all of the MATLAB variables stored from the current workspace, which essentially means that all the variables you defined. However, you may also specify precisely which variable(s) to clear by using the clear command followed by the variable names. For example, clear x would clear the variable x from memory, and clear myVar would clear the variable myVar from memory. Note that clearing a symbolic variable is application-specific and <u>not</u> mandatory.

How can I convert from syms data type to other data types?

Since `syms` is a data type in MATLAB, it often needs to be converted to a different data type, such as double or char, to display it. See Example 2 for some specific use cases. Using the wrong conversion could result in an error. Some commonly used functions to convert the `syms` data type to other data types/formats are:

- `double()` - > Convert a symbolic matrix with numerical value to MATLAB numeric data type.
- `char()` - > Convert symbolic objects to strings.

Example 2

Input the function $f(s) = s^2 + 4s + 56$ to MATLAB, convert it from sym data type to char data type. Then output the converted function using fprintf().

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To replace a symbolic variable with a value
fprintf('To replace a symbolic variable with a value.\n\n')
```

MATLAB Code (continued) example2.m

```
%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
syms s                  %Defining 's' as symbolic variable
f = s^2 + 4*s + 56      %Defining function with symbolic variable 's'
typeFunc = class(f);    %Checking the data type of the function
fprintf('The data type of the function is currently %s.\n\n',typeFunc)

%------------------------------ SOLUTION -----------------------------
strEq = char(f);              %Converting the function to a string data type
typeConvert = class(strEq);   %Checking the data type of the converted
function
fprintf('The data type of the function is currently %s.\n\n',typeConvert)

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The function we input is f(s) = %s.\n\n',strEq)
```

Command Window Output Exam

```
PURPOSE
To replace a symbolic variable with a value.

INPUTS

f =

s^2 + 4*s + 56

The data type of the function is currently sym.

The data type of the function is currently char.

OUTPUTS
The function we input is f(s) = 4*s + s^2 + 56.
```

Can I replace a symbolic variable with a value?

MATLAB makes changing/substituting the value or the name of a symbolic variable simple. Using the `subs()` function, you can assign a value for the symbolic variable (e.g., $x = 1$) or rename it (e.g., $x = a$). This function is handy when we plot equations later in this lesson and in future lessons.

Example 3

Find the value or form of the function $f(s) = s^2 + 4s + 56$ at $s = a$ and $s = 1$.

Solution

MATLAB Code example3.m

```matlab
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To replace a symbolic variable with a value or character
fprintf('To replace a symbolic variable with a value or character.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
syms s                   %Defining 's' as symbolic variable
f     = s^2 + 4*s + 56;  %Defining function with symbolic variable 's'
strEq = char(f);         %Converting the function to a string data type
fprintf('The function we input is f(s) = %s.\n\n',strEq)

%------------------------------ SOLUTION ------------------------------
f1 = subs(f,s,1);   %Finding f(1) using subs() to substitute a number
syms a              %Defining 'a' as a symbolic variable. This could have
                    %    been done when we defined 's', but we have kept
                    %    them separate for clarity.
fa  = subs(f,s,a);  %Replacing 's' with 'a' using subs()

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The value of f(1) is %g.\n',f1)
fprintf('f(a) = %s.\n',char(fa))
```

Command Window Output Example 3

```
PURPOSE
To replace a symbolic variable with a value or character.

INPUTS
The function we input is f(s) = 4*s + s^2 + 56.

OUTPUTS
The value of f(1) is 61.
f(a) = 4*a + a^2 + 56.
```

How can I change the output format of syms?

You can change the output type (scientific, floating-point, etc.) directly within an `fprintf()` function if your number is in decimal form. However, MATLAB returns exact solutions by default; that is, it will return "1/3" instead of "0.33333…". The precision of an answer can be adjusted using the function `vpa()`.

This is sufficient for most purely numerical answers. If you have an answer that contains symbolic variables like x, t, etc., you may need to use the function `simplify()` to help clean up and simplify messy solutions.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Create a symbolic variable	`syms`	`syms x`
Convert a variable (with numeric value) to double precision (numeric data type)	`double( )`	`double(var)`
Convert a variable to the char data type	`char( )`	`char(var)`
Substitute a new value or variable for the current symbolic variable	`subs( )`	`subs(eq,replaceVar,newVar)`
Adjust the precision of a numeric variable	`vpa( )`	`vpa(x,3)`
Simplify an algebraic expression	`simplify( )`	`simplify (x)`

Multiple Choice Quiz

(1). If one was going to write $y = 3 * x$ in an m-file, the variable(s) that would need to be declared as symbolic is (are)

(a) `x`

(b) `y`

(c) both `x` and `y`

(d) either `x` or `y`

(2). Given the following code:

```
clc
clear

m = 7;
syms m c
e = m*c^2;
e
```

the output of the **last line** in the Command Window is

(a) `e = 7*c^2`

(b) `e = m*c^2`

(c) `e = 14m`

(d) `Undefined function or variable…`

(3). To display the function $f(x) = x^2$ in the Command Window given the program

```
clc
clear

syms x
f = x^2;
```

the correct line of code to add is

(a) `fprintf('The function is f(x) = %g.\n',char(f))`

(b) `fprintf('The function is f(x) = %s.\n',char(f))`

(c) `fprintf('The function is f(x) = %g.\n',double(f))`

(d) `fprintf('The function is f(x) = %f.\n',vpa(f))`

(4). The symbolic variable that will cause an error when used in an m-file is

(a) `ex`

(b) `sin`

(c) `tt`

(d) `x`

(5). The Command Window output of the **last line** of this program is

```
clc
clear

y=5
y=3*x
```

(a) `3x`

(b) `15`

(c) `y`

(d) `Undefined function or variable 'x'.`

Problem Set

(1). In a single m-file, display the expression in all the parts below in the Command Window.

(a) $y = \frac{2}{3}x^2 + x + x^{\frac{1}{2}}$

(b) $z = \frac{x^2 y}{x+1}$

(c) $P = \frac{mRT}{V}$

(d) $x = \frac{M_\nu}{M_\nu + M_l}$

(2). In most cases, the weight w of an object is determined by the mass of the object m and the acceleration acting on the body of the object a given by the relationship

$$w = ma$$

Display the general expression to find the weight of a body.

(3). Evaluate the following functions. Output both the function and the resulting value from evaluating the functions at the given numbers to the Command Window. You should use a single `fprintf()` function for each function-value pair of outputs.

(a) Evaluate $f(x) = x^4 + 98$ when $x = 2$.

(b) Evaluate $G(s) = s^2 + 5s - 6$ when $s = 3.4$.

(c) Evaluate $P(l) = l^2 + 3$ when $l = 8.1$.

(4). The brake power of an internal combustion engine is found by

$$P = r * T$$

where, P = Power (Watt) r = Rate (radian/sec) T = Torque (N-m) and the engine torque is given by

$$T = 0.62r$$

Using MATLAB,

(a) display the general expression for the brake power generated by the engine,

(b) find the value of the brake power when r is 350 rad/sec,

(c) plot power vs. rate with values of r from 0 to 630 rad/sec. Use appropriate labels, title, etc., as required to describe the plot.

(5). The acceleration of a point on a body is composed of four components: the tangential, normal, sliding, and Coriolis. Assuming that sliding and Coriolis effects of acceleration are zero, the tangential $ (a_tan) $ and normal (a_{normal}) components of acceleration can be combined as functions of time, t,

$$a_{total} = \sqrt{a_{\tan}^2 + a_{normal}^2}$$

where

$$a_{\tan} = 2\sin\left(\frac{1}{2}t\right) + t^2$$

$$a_{normal} = \frac{\left(\frac{1}{3}t^3 - 4\cos(\frac{1}{2}t)\right)^2}{300}$$

The units of time are seconds and the units of acceleration are m/s^2. Display the equation for the total acceleration of the body in the Command Window (use t as a symbolic variable). Find the value of the acceleration of the body when $t = 4.3$ seconds.
Hint: Do not name the variable for the tangential acceleration "`atan`". It is a predefined function in MATLAB.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.3: Solution of Linear and Nonlinear Equations

Learning Objectives

After reading this lesson, you should be able to:

- *solve linear equations in MATLAB,*
- *solve nonlinear equations in MATLAB,*
- *manipulate polynomials and polynomial coefficients,*
- *plot data with the syms data type.*

How do I solve for roots of a linear equation?

In this section, we will go over using MATLAB to solve for the roots of an equation using `solve()`. For example, $x - 2 = 0$ has a root at $x = 2$.

Example 1 shows the procedure for solving for the root of a linear equation in MATLAB. Note that when the equation is defined in Example 1, we write `y = 9*x +7 == 10`. This means that the equation `9*x + 7 == 10` is stored in the variable y, which we call subsequently in the program like we would any other variable. The right-hand side of the equation that you enter does not have to be equal to zero. For example, you could also write `9*x == 3` and still get the same answer from MATLAB.

You might be asking at this point, "Why do we need double equals when entering the equation?" The simple answer to this is that MATLAB needs to recognize that you are entering an equation rather than storing something in a variable. Remember that y = `9*x + 7 == 10` has two parts. Entering the equation `9*x + 7 == 10` is stored in the variable y. MATLAB needs a way to differentiate between these two operations. All you need to remember is to put the double equals (==) when entering an equation. MATLAB will return an error if you fail to do so.

Example 1

Solve for the root of the $9x + 7 = 10$.Display the equation and the root of the equation using a single `fprintf()` function.

Solution

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find the root of a linear equation
fprintf('To find the root of a linear equation.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
syms x                 %Defining 'x' as symbolic independent varia
y    = 9*x + 7 == 10;  %Defining equation with symbolic variable '
strEq = char(y);       %Converting the equation to a string data t
fprintf('The equation to solve is %s = 0.\n\n',y)

%------------------------------ SOLUTION -----------------------------
soln  = solve(y,x);    %Solving for roots of the equation
soln  = double(soln);  %Converting syms variable "soln" to double 
                       %    type

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The solution to %s = 0 is x = %.4g.\n\n', strEq, soln)
```

```
Command Window Output                                          Example 1

PURPOSE
To find the root of a linear equation.

INPUTS
The equation to solve is 9*x + 7 == 10 = 0.

OUTPUTS
The solution to 9*x + 7 == 10 = 0 is x = 0.3333.
```

What is a nonlinear equation?

A nonlinear equation is an equation that has nonlinear terms of the unknowns. One of the simplest examples of a nonlinear equation is the quadratic equation that has the form

$ax^2 + bx + c = 0$ (1)

where,

$a \neq 0$.

The values of a, b, and c are constant coefficients and x is the independent variable. If one were to solve a nonlinear equation using traditional methods they would need to be provided at least two items: the nonlinear equation and the unknown variable to solve for.

Take the case of the equation given in the above Equation (1). The nonlinear equation is a quadratic equation and the variable to solve for is x. This example can be solved using the widely known solution of a quadratic equation as

$x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ (2)

However, other more complex nonlinear equations can take longer to solve exactly, while most nonlinear equations are impossible to solve analytically. Another example of a nonlinear equation is $\sin(2x) = \dfrac{3x}{7}$. In these cases, MATLAB can be used to solve the nonlinear equations or even to verify the results obtained from an analytical method.

How can I use MATLAB to solve nonlinear equations?

As we know, MATLAB is a tool. Just like a mechanic might use a wrench to solve a problem, an engineer, scientist or mathematician might use MATLAB to solve a problem. In the case of solving a nonlinear equation, MATLAB can be used to solve nearly any equation (limited more so by the operating computer than the program itself).

We used `solve()` to find the roots of a linear equation as a demonstration of symbolic variables. However, `solve()` is also used to find the roots of nonlinear equations. In this case, we may get multiple roots. Unique, none and infinite number of solutions are other possibilities for nonlinear equations. Multiple roots will be stored in vector form, which you can see in Example 2. The variable `soln` is a vector that holds the three roots we found from the third-order polynomial equation.

You can see in Example 2 that the solution to the nonlinear equation has three possible values. We know this is correct because the equation is a cubic equation. The outputs to the solve command are stored in a vector and can each be pulled out of the vector by using matrix references, which we covered in Lesson 2.6.

Example 2

Solve the nonlinear equation $x^3 - 15x^2 + 47x = 33$ for x using MATLAB (that is, find its roots).

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To find the root of a nonlinear equation
fprintf('To find the root of a nonlinear equation.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
syms x                          %Defining 'x' as symbolic variable
y = x^3 - 15*x^2 + 47*x == 33;  %Defining equation
strEq = char(y);                %Converting equation to string data
                                %    type
fprintf('The equation to solve is %s.\n\n',y)

%------------------------------ SOLUTION ------------------------------
soln = solve(y,x);    %Solving for roots of the equation
soln = double(soln);  %Converting syms variable "soln" to double data
                      %    type

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The solution to %s is\n',strEq)
fprintf('x1 = %g, x2 = %g, and x3 = %g.\n\n',soln(1),soln(2),soln(3))
```

Command Window Output Example 2

```
PURPOSE
To find the root of a nonlinear equation.

INPUTS
The equation to solve is 47*x - 15*x^2 + x^3 == 33.

OUTPUTS
The solution to 47*x - 15*x^2 + x^3 == 33 is
x1 = 1, x2 = 3, and x3 = 11.
```

MATLAB also provides a method to solve an equation numerically using `vpasolve()` (note that solve() uses analytical methods). Although polynomial equations have a finite number of solutions (which is equal to the order of the polynomial), other nonlinear equations can possibly have an infinite number of solutions (e.g., $\tan(x) = x$). Therefore, MATLAB does not report all solutions in the same way. Check the `vpasolve()` documentation for specifics.

In Example 3, we show an application that shows a real-world problem. The only difference here is the prerequisite mathematical knowledge to set up the problem (find the appropriate equation). The programming portion is more or less the same, which is the beauty of MATLAB and many other similar programming languages. Once written, a program like the one in Example 2 can be quickly edited for a completely different equation like the one in Example 3. We chose to use the aforementioned `vpasolve()` in Example 3.

Example 3

Find the depth x to which a ball is floating in water (see Figure 1) based on the following cubic equation

$$4R^3S - 3x^2R = -x^3$$

where,
R = radius of the ball,

S = specific gravity of the ball.

Use values of R and S to be 0.055 and 0.6, respectively.

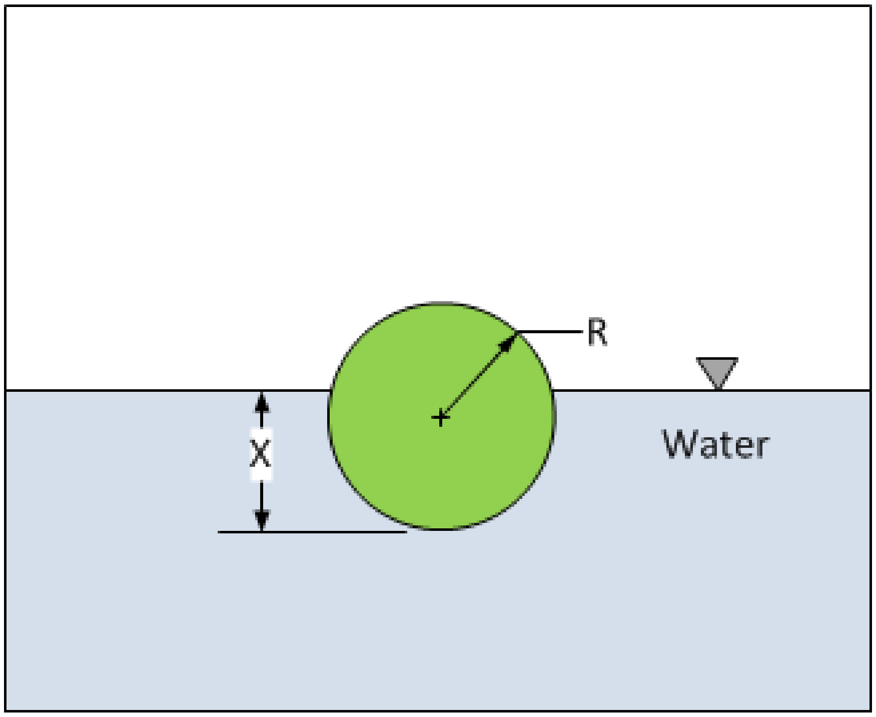


Figure 1: Diagram of a partially submerged sphere in water for use with Example 3.

Solution

MATLAB Code example3.m

```matlab
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find the depth to which a floating ball is submerged in water
fprintf(['To find the depth to which a floating ball is submerged ',...
         'in water.\n\n'])
```

MATLAB Code (continued) examp

```
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
R = 0.055;                         %Radius of ball
S = 0.6;                           %Specific gravity of the ball

syms x                             %Defining 'x' as symbolic variabl
eq = 4*R^3*S - 3*x^2*R == -x^3;    %Defining equation
precisionEq = vpa(eq,5);           %Adjust the precision of "eq"
strEq = char(precisionEq);         %Converting equation to a string

fprintf('The equation to solve is %s.\n\n',eq)

%------------------------------ SOLUTION ----------------------------
soln = vpasolve(eq, x);  %Numerically solving for roots of the equ
soln = double(soln);     %Converting syms variable "soln" to doubl
                         %    data type

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The solution to %s is\n',strEq)
fprintf('x1 = %g, x2 = %g, and x3 = %g.\n\n',soln(1),soln(2),soln(
```

Command Window Output Exam

```
PURPOSE
To find the depth to which a floating ball is submerged in water.

INPUTS
The equation to solve is 7365784908632225/18446744073709551616 -
(33*x^2)/200 == -x^3.

OUTPUTS
The solution to 0.0003993 - 0.165*x^2 == -1.0*x^3 is
x1 = -0.0437371, x2 = 0.0623776, and x3 = 0.14636.
```

Note the three possible solutions to the third order polynomial equation for Example 2. Since the limits of x are given by $0 \leq x \leq 2R$ and $R = 0.055$, the limits of x are $0 \leq x \leq 0.11$. Hence, the only physically acceptable solution is $x = 0.0624$.

Is there a faster way to work with polynomial equations in MATLAB?

We can create a symbolic polynomial from a vector of coefficients using the function `poly2sym()`. (You can do the reverse operation with `sym2poly()`.) This can be especially helpful when using functions that return a vector of coefficients directly such as `polyfit()`, which we will cover later in Lessons 4.7 and 4.8. You can see in Example 4 that the coefficient output (`coefs`) is not very intuitive. However, we can directly convert the coefficients to polynomial form, which is more intuitive, and find the roots of the corresponding polynomial with the `roots()` function.

Example 4

Given any size vector of polynomial coefficients, generate the full polynomial in symbolic form and find its roots.

Test the program using the coefficient vector $[0.003 \quad -0.0748 \quad -4.2660 \quad 1.8370]$.

Solution

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ----------------------------
fprintf('PURPOSE\n')
%To work with coefficients of a polynomial equation
fprintf('To work with polynomial coefficients.\n\n')

%------------------------------ INPUTS -----------------------------
fprintf('INPUTS\n')
coefs = [0.0030, -0.0748, -4.2660, 1.8370]  %Coefficients of an
                                             %   arbitrary polynomi

%------------------------------ SOLUTION ---------------------------
f = poly2sym(coefs);  %Converting coefficients directly to symboli
approxF = vpa(f,4);   %Adjusting precision

soln = roots(coefs);  %Finding the roots directly from coefficient

%------------------------------ OUTPUTS ----------------------------
fprintf('OUTPUTS\n')
fprintf('The polynomial equation is %s == 0\n\n',char(approxF))
fprintf('Its roots are:\n')
disp(soln)
```

```
Command Window Output                                              Example 4

PURPOSE
To work with polynomial coefficients.

INPUTS

coefs =
    0.0030   -0.0748   -4.2660    1.8370

OUTPUTS
The polynomial equation is 0.003*x^3 - 0.0748*x^2 - 4.266*x + 1.837 == 0

Its roots are:
   52.0350
  -27.5291
    0.4275
```

Can I plot with symbolic variables?

Plotting with `syms` has all the same considerations as any other function you wish to plot with the additional step of converting to a data type that is accepted by the function you are using to plot (e.g., `plot()`, [`bar()`](https://www.mathworks.com/help/matlab/ref/bar.html?s_tid=srchtitle), etc.). That is, you will likely need to convert from `sym` to a `numeric` data type to plot the data.

To get a vector of data points to plot, we can use the `subs()` function. Although it was used in the previous examples to replace a variable with just a single number or character, it also accepts vector inputs as the replacement value(s). When the input is a vector, the output is a corresponding vector as well.

Example 5

Plot the function $y(x) = 4x^3 - 20x^2 + 9x$ and its zeros on the same plot. Choose the domain to plot the function based upon the solutions found for the zeros of the function. Include title, axis labels, legend, and plot grid.

Solution

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ----------------------------
%To plot a function and its roots.

%------------------------------ INPUTS -----------------------------
syms x                     %Defining 'x' as symbolic indep. variab
y = 4*x^3 - 20*x^2 + 9*x;  %Defining function with symbolic variab

%------------------------ SOLUTION/OUTPUTS -------------------------
soln = solve(y, x);                  %Solving for zeros of the fu

xPoints = min(soln):0.1:max(soln);   %Defining domain to plot fun
yPoints = subs(y,x,xPoints);         %Find the value of the equat
                                     %    each 'x' point
figure                               %Creating a blank figure
plot(soln,zeros(length(soln),1),'*','MarkerSize',8) %Plotting the
%Note y coordinates will be zero.
hold on                              %Telling MATLAB to plot on s
                                     %    plot
grid on                              %Placing grid on the plot
plot(xPoints, yPoints,'r')           %Plotting function y(x)

%Figure formatting (see lesson m-file for full code)
title('Equation and Its Roots')
xlabel('Independent Variable, x')
ylabel('Dependent Variable, y')
legend('Function Roots','Function, y(x)')
```

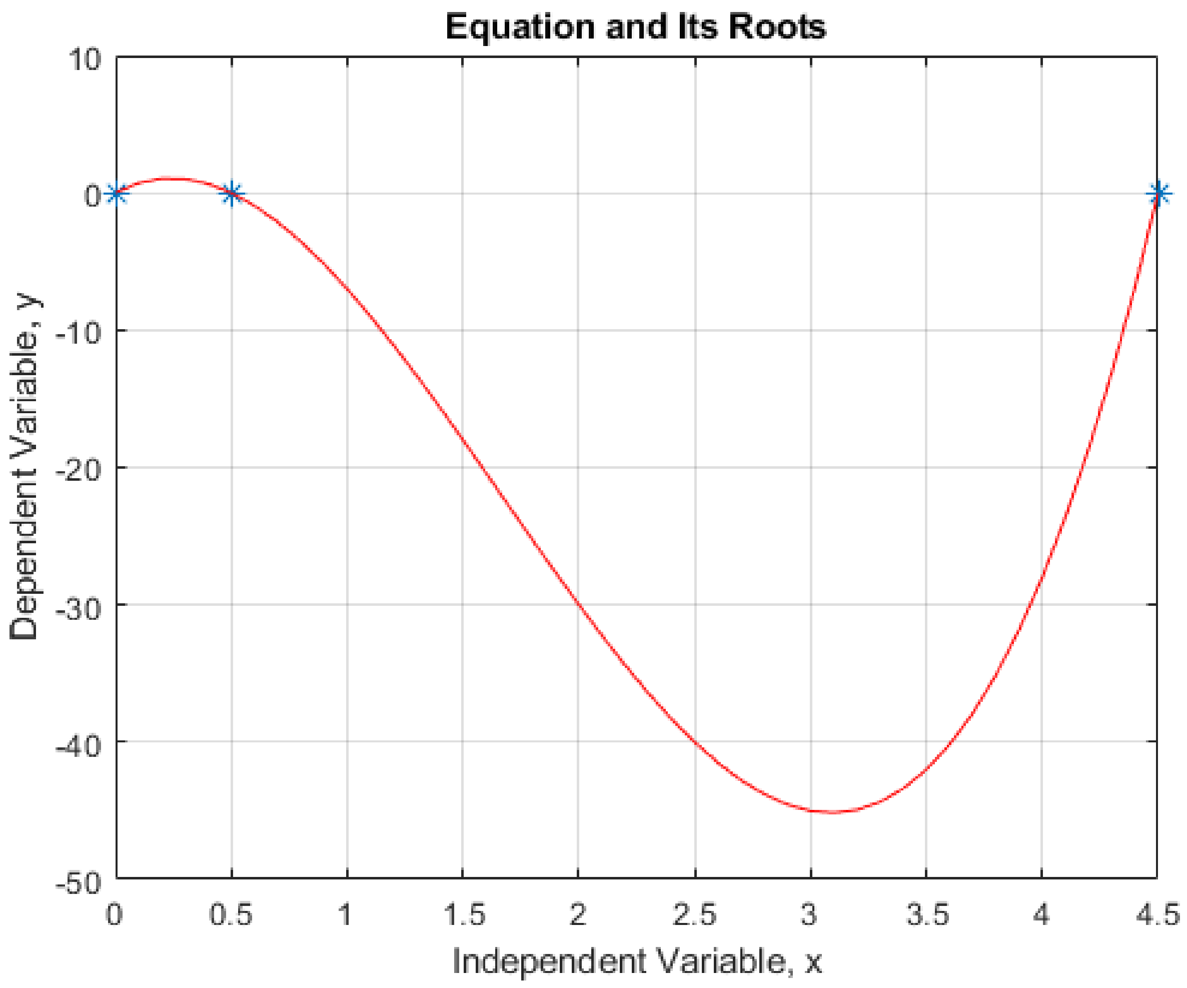


Figure 6: The MATLAB figure output for Example 5.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Find the roots of an equation	`solve()`	`solve(eq,x)`

Task	Syntax	Example Usage
Numerically find the roots of an equation	`vpasolve()`	`vpasolve(eq,x)`
Form a symbolic polynomial function from coefficients	`poly2sym()`	`poly2sym(coefs,x)`
Extract the coefficients from a polynomial	`sym2poly()`	`sym2poly(polynomial)`
Get the roots of a polynomial using only is coefficients as an input	`roots()`	`roots(coefs)`

Multiple Choice Quiz

(1). A MATLAB function for solving an equation analytically is

(a) `fsolved()`

(b) `nonlinear()`

(c) `vpasolve()`

(d) `solve()`

(2). The MATLAB function to find the solution to a polynomial equation given only its coefficients is

(a) `solve()`

(b) `coefs()`

(c) `roots()`

(d) `polysolve()`

(3). To solve $x^2 + 2x = 0$, which line of code should be added to the program

```
clc
clear

syms x
```

(a) `solve(x^2+2x==0,x)`

(b) `solve(x^2-2x,x)`

(c) `solve(x^2+2*x=0,x)`

(d) `solve(x^2+2*x==0,x)`

(4). The MATLAB substitution function `subs()` has _______________ input variables

(a) `1`

(b) `2`

(c) `3`

(d) `4`

(5). The output solutions for a single nonlinear equation given by the `solve()` function

(a) gives only one solution to the equation.

(b) gives only the physically acceptable solution(s).

(c) provides the solutions as separate outputs.

(d) stores solutions in a single vector.

Problem Set

(1). Use the `solve()` function to find the solution of the equation $3x^2 + 2x = 5$. Display the answers in the Command Window.

(2). Given that the value of the two variables a and b can be changed, set up the equation $ax^2 + 2x = b$ so that it can be solved for any real values of a and b.

(3). Solve the nonlinear equation $2x + e^{-x} = 5$. Output both the original equation and at least one of its roots using a single `fprintf()` function.

(4). You are working for 'DOWN THE TOILET COMPANY Inc.' that makes floats for H3H3 toilets. The floating ball has a specific gravity of 0.6 and has a radius of 5.5 cm. You are asked to find the depth to which the ball is submerged when floating in water (Figure A).

The equation that gives the depth x in meters to which the ball is submerged underwater is given by

$$x^3 - 0.165x^2 + 3.993 \times 10^{-4} = 0$$

Find the depth, x, to which the ball is submerged underwater.

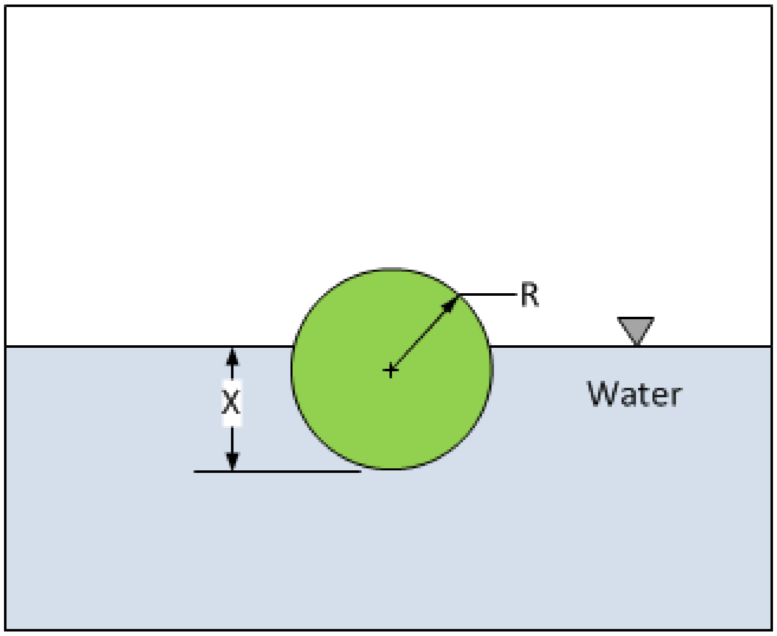


Figure A: Floating ball in water for Exercise 4.

(5). You have a spherical storage tank containing oil (Figure B). The tank has a diameter of 6 ft. You are asked to calculate the height, h, to which a dipstick 8 ft long would be wet with oil when immersed in the tank when it contains 4 ft^3 of oil. The equation that gives the height, h, of the liquid in the spherical tank for the above-given volume and radius is given by

$$f(h) = h^3 - 9h^2 + 3.8197 = 0$$

Find the height, h, to which the dipstick is wet with oil.

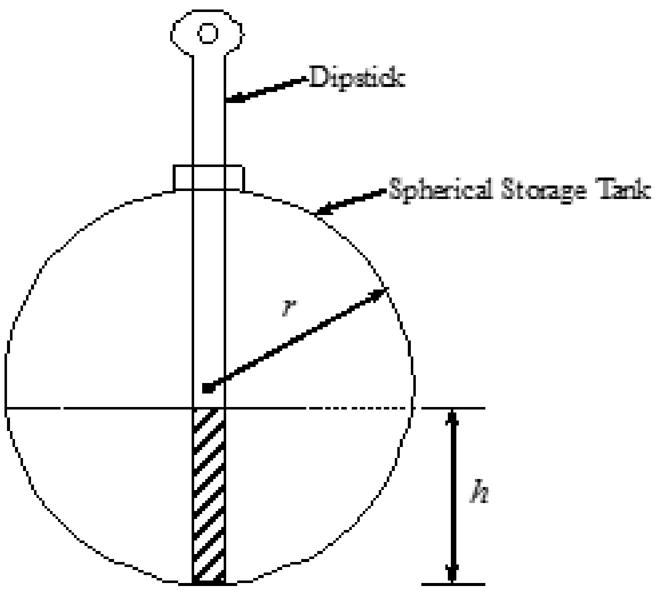


Figure B: Dipstick inside spherical tank which contains some oil. Used in Exercise 5.

(6). You are making a cool bookshelf to carry your interesting books that range from $8\frac{1}{2}$" to 11" in height. The bookshelf material is wood, which has a Young's Modulus of 3.667 Msi, length (L) of 29", a thickness of 3/8", and width of 12". You want to find the maximum vertical deflection of the bookshelf. The vertical deflection of the shelf is given by

$$v(x) = 0.42493 \times 10^{-4}x^3 - 0.13533 \times 10^{-8}x^5 - 0.66722 \times 10^{-6}x^4 - 0.018507x$$

where x is the position along the length of the beam (Figure C). To find the maximum deflection we need to find where

$$f(x) = \frac{dv}{dx} = 0$$

and conduct the second derivative test. The equation that gives the position,x, where the deflection is extreme (minimum or maximum) is given by

$$-0.67665 \times 10^{-8}x^4 - 0.26689 \times 10^{-5}x^3 + 0.12748 \times 10^{-3}x^2 - 0.018507 = 0$$

(a) Plot the deflection of the beam, $v(x)$, for values of x from 0 to L.

(b) Find the position x where the deflection is maximum.

(c) Find the value of the maximum deflection.

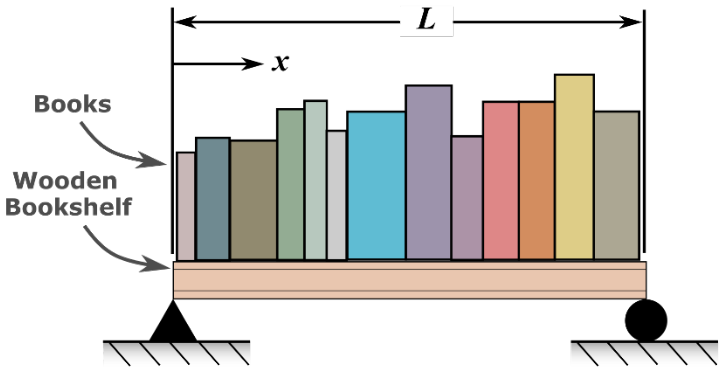


Figure C: Schematic of bookshelf used in Exercise 6.

(d) A trunnion has to be cooled from a temperature of $80^{\circ}F$before it is shrink-fitted into a steel hub (See Figure D on the next page). The equation that gives the temperature, T_f, to which the trunnion has to be cooled to obtain the desired contraction is given by

$$f(T_f) = -0.50598 \times 10^{-10} T_f^3 + 0.38292 \times 10^{-7} T_f^2 + 0.74363 \times 10^{-4} T_f + 0.88318 \times 10^{-2} = 0$$

Find the roots of the equation to find the temperature, T_f, to which the trunnion has to be cooled. Choose the physically acceptable root of the equation.

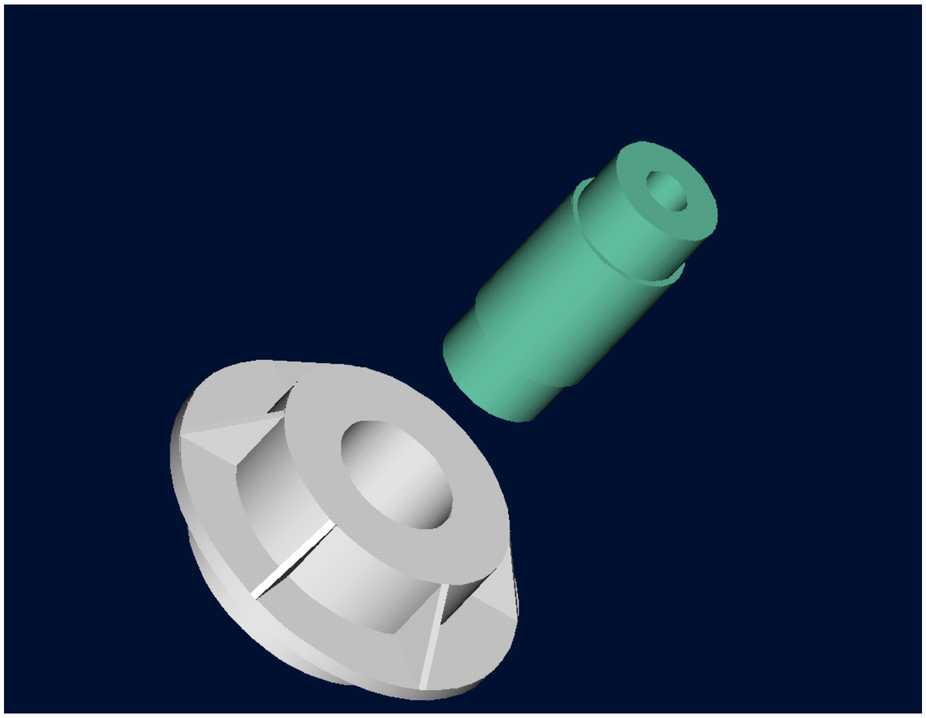


Figure D: Trunnion and hub shown prior to shrink fitting for Exercise 7.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.4: Differential Calculus

Learning Objectives

After reading this lesson, you should be able to:

1) *find the derivatives of continuous functions,*
2) *find the derivatives of discrete functions,*
3) *use derivatives to solve basic engineering problems.*

What is a derivative?

The derivative of a function represents the rate of change of a variable with respect to another variable (see Figure 1).

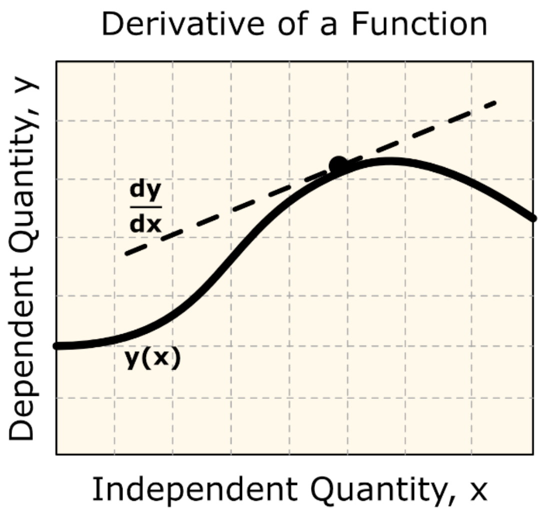


Figure 1: Shown is a line tangent to a function, $y(x)$.

For example, the velocity of a body is defined as the rate of change of the location of the body with respect to time. The location is the *dependent* variable while time is the *independent* variable. Now if we measure the rate of change of velocity with respect to time, we get the acceleration of the body. In this case, the velocity is the *dependent* variable while time is the *independent* variable.

Recall from calculus, the derivative of a function $f(x)$ is defined as

$$f^{'}(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

The value of the first derivative of a function evaluated at a point on the function is the slope of the line tangent to the function at that point. This concept is depicted in Figure 1.

How do I take the derivative of a function in MATLAB?

To do the differentiation, one needs three inputs:

1. The function *f(x)* that needs to be differentiated.
2. The variable with respect to which the function needs to be differentiated, x.

3. The order of derivative needed, n.

In MATLAB, these three inputs are required to find the derivative of a symbolic function. To differentiate a symbolic function, the MATLAB function `diff()` can be used. The order of the derivative can be specified, as shown in Example 1, but the `diff()` function defaults to a first-order derivative if no order is given.

It is important to note that the `diff()` function is also used to find the difference between corresponding points in two vectors (covered in more detail later in this lesson), and if the programmer does not use the correct input variable placement (syntax), the output would be incorrect. Example 1 shows the `diff` function being used to find a derivative.

Example 1

Using the `diff()` function, write a program that finds and outputs the following.

(a) $\frac{d}{dx}\left(7e^{3x}\right)$

(b) $\frac{d^2}{dx^2}\left(\sin(x^2)+x^6\right)$

Solution

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To differentiate symbolic functions
fprintf('To differentiate symbolic functions.\n\n')
```

```
MATLAB Code (continued)                                          example1.m
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
syms x             % Define x as a symbolic variable
yA = 7*exp(3*x);   % A) function to be differentiated
yB = sin(x^2)+x^6; % B) function to be differentiated

fprintf('The function in part (A) is %s.\n', char(yA))
fprintf('The function in part (B) is %s.\n\n', char(yB))

%----------------------------- SOLUTION ----------------------------
%Finding the derivatives
dydxA = diff(yA,x,1);    %Part (A) first derivative
dydxB = diff(yB,x,2);    %Part (B) second derivative

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('A)  The first derivative of %s is equal to\n',char(yA))
fprintf('     %s.\n\n',char(dydxA))
fprintf('B)  The second derivative of %s is is equal to\n',char(yB))
fprintf('     %s.\n\n',char(dydxB))
```

To solve the problem given in Example 3 one must first enter the symbolic functions to be differentiated. The **syms** command must then be used to define the symbolic variable(s) *before* defining any function(s).

```
Command Window Output                                          Example 1
PURPOSE
To differentiate symbolic functions.

INPUTS
The function in part (A) is 7*exp(3*x).
The function in part (B) is sin(x^2) + x^6.

OUTPUTS
A)  The first derivative of 7*exp(3*x) is equal to
     21*exp(3*x).

B)  The second derivative of sin(x^2) + x^6 is is equal to
     2*cos(x^2) - 4*x^2*sin(x^2) + 30*x^4.
```

Example 2

Find the slope of the tangent line to the function$s(t) = e^t \sin(2t)$ at $t = 3$, using the `diff()` and `subs()` functions.

Solution

The `diff()` function is used to find the slope of the tangent line (first derivative) and then the `subs()` function is called to evaluate the slope of the tangent line at the specified value of t. To show the inputs and outputs for this example, the `char()` function is used to convert symbolic expressions into printable strings.

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find a derivative of a function and evaluate it at a given value
fprintf('To find a derivative of a function\n')
fprintf('and evaluate it at a given value.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
syms t                    %Define t as a symbol
s = exp(t)*sin(2*t);      %Defining the function to be differentiated

fprintf('The function to be differentiated is %s.\n\n',char(s))

%----------------------------- SOLUTION ----------------------------
dsdt = diff(s,t,1);          %Finding the first derivative
time = 3;                    %What to replace 't' with.
                             %    We cannot name this variable "t"!!
dsdt3 = subs(dsdt,t,time);   %Finding the value of dydx at t=3

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The derivative of the function is %s.\n',char(dsdt))
fprintf('The value of s''(%g) = %g.\n',time,double(dsdt3))
```

Command Window Output Example 2

```
PURPOSE
To find a derivative of a function
and evaluate it at a given value.

INPUTS
The function to be differentiated is sin(2*t)*exp(t).

OUTPUTS
The derivative of the function is 2*cos(2*t)*exp(t) + sin(2*t)*exp(t).
The value of s'(3) = 32.9589.
```

Where are derivatives used in engineering?

Derivatives have many uses in engineering and mathematics. Some derivatives are easy to find, others are complex, and some are even partial.

Derivatives are used to set up differential equations, which will be covered in Lesson 4.9. In engineering and physics, one of the most common uses of derivatives is in the relationship of position, velocity, and acceleration functions of a body. Where given the position of a body as a function of time, the derivative of position with respect to time is the velocity function. The derivative of velocity with respect to time is the acceleration function of the body.

Another common use of derivatives is finding the location of minimum and maximum points of a function. We know that the value of the derivative is the slope of the line tangent to a function at a given point. If we set the found derivative equal to zero (tangent slope is zero) and solve for the independent variable, we can find the local minima or maxima of a function. This technique is used for optimization.

Example 3

The position of a body is defined by the function *s(t)* as

$$s(t) = 7t^3 + \ln(2t) - 9.8t^2$$

where t is in seconds, and s is in m/s. Using MATLAB, output the velocity and acceleration functions of the body. Also, find the velocity and acceleration of the body at $t = 2.5$ seconds.

Solution

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To find the vel and acc at a given time given s(t)
fprintf('To find the vel and acc at a given time given s(t).\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
syms t                            %Defining t as a symbolic variable
s = 7*t^3 + log(2*t) - 9.8*t^2;   %Defining the position function, s(t)

fprintf('The position function is %s.\n\n',char(s))
```

MATLAB Code (continued) example3.m

```
%------------------------------ SOLUTION -----------------------------
vel     = diff(s,t,1);        %First derivative of pos to get vel
acc     = diff(s,t,2);        %Second derivative of pos to get acc
time    = 2.5;                %Time we want to find vel and acc
velTime = subs(vel,t,time);   %Velocity at time=2.5s
accTime = subs(acc,t,time);   %Acceleration at time=2.5s

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The velocity function is %s.\n',char(vel));
fprintf('The acceleration function is %s.\n',char(acc))
fprintf('The velocity at t=%gs is %g m/s.\n',time,double(velTime))
fprintf('The acceleration at t=%gs is %g m/s^2.\n',time,double(accTime))
```

In Example 3, the `diff()` function is used to find the velocity and the acceleration of the body. The `subs()` function is then used to find the velocity of the rocket at $t = 2.5$ seconds.

```
Command Window Output                                    Example 3

PURPOSE
To find the vel and acc at a given time given s(t).

INPUTS
The position function is log(2*t) - (49*t^2)/5 + 7*t^3.

OUTPUTS
The velocity function is 1/t - (98*t)/5 + 21*t^2.
The acceleration function is 42*t - 1/t^2 - 98/5.
The velocity at t=2.5s is 82.65 m/s.
The acceleration at t=2.5s is 85.24 m/s^2.
```

How do I find the derivative of a discrete function in MATLAB?

The `diff()` function can also be used in the case of taking the derivative of a function with discrete data points. That is, differentiating a matrix or vector containing independent variables such as $[2 \quad 4 \quad 6 \quad 8]$.

If a continuous/exact mathematical function is not available and a discrete data set is presented (e.g., position sensor reading), the `diff()` function can be used to find the differences, or steps, between subsequent data entries which in turn can be used to find the numerical derivative.

The following examples show this discrete/numerical differentiation performed with the `diff()` function. The difference between subsequent vector elements is used to find the numerical derivative. This approximation is generally referred to as the finite difference formula.

Example 4

Find the approximate derivative of the discrete function, *F(t)*, given the following points.

The discrete function *F(t)* is given by

t	0.0	0.5	1.0	1.2	2.0	2.5	3.0
F	0.0	10	20	45	68	88	100

Use the forward divided difference method as the numerical algorithm to estimate the first derivative between the discrete data points.

The forward divided difference formula is given by $\frac{\mathrm{dF(t)}}{\mathrm{dt}} \approx \frac{\mathrm{F\,(t + \Delta t) - f(t)}}{\mathrm{\Delta t}}$.

Solution

```
MATLAB Code                                                    example4.m
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To approximate the derivative from discrete data
fprintf('To approximate the derivative from discrete data.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
t = [0  0.5  1.0  1.5  2.0  2.5  3.0];  %Data of independent variable
                                        %    vector t
F = [0  10  20  45  68  88  100] ;      %Data of dependent variable
                                        %    vector F
disp('t =')
disp(vpa(t,2))
disp('F =')
disp(vpa(F,2))

%------------------------------ SOLUTION ------------------------------
dt   = diff(t);  %Finding difference between elements in t
dF   = diff(F);  %Finding difference between elements in F
dFdt = dF./dt;   %Differentiating at each discrete point

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
disp('Approximate value of derivatives between each pair of')
disp('given data points:')
disp(vpa(dFdt,2))
```

Note, the period (.) in dF./dt acts as an element-by-element array operator. This is fully covered in Lesson 4.6 on Linear Algebra, which you can review as needed.

```
Command Window Output                                          Example 4
PURPOSE
To approximate the derivative from discrete data.

INPUTS
t =
[ 0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]

F =
[ 0, 10.0, 20.0, 45.0, 68.0, 88.0, 100.0]

OUTPUTS
Approximate value of derivatives between each pair of
given data points:
[ 20.0, 20.0, 50.0, 46.0, 40.0, 24.0]
```

Note that other numerical differentiation algorithms are possible and should be considered depending on the particular discrete data set and application. In general, as the data sample resolution and sampling rate increases, the differences between the values obtained using numerical differentiation methods becomes negligible.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Find the derivative of a symbolic function	`diff()`	`diff(func,x,2)`
Find the difference between adjacent elements in a vector	`diff()`	`diff(xPoints)`

Multiple Choice Quiz

(1). The MATLAB function for symbolic differentiation is

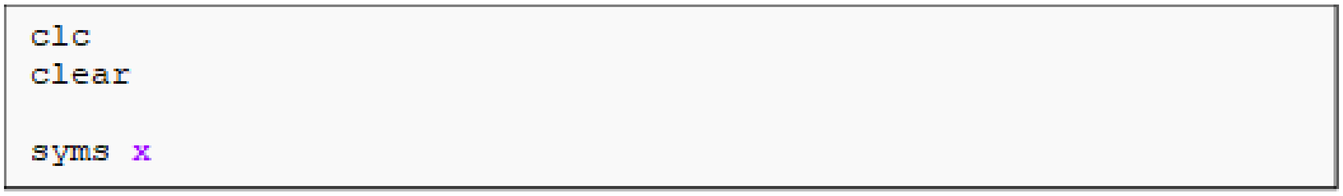
(a) `dif()`

(b) `diff()`

(c) `int()`

(d) `differentiate()`

(2). To find $\frac{d^2}{dx^2}\left(7e^{4x}\right)$, the correct line of code to add to the following program is



```
clc
clear

syms x
```

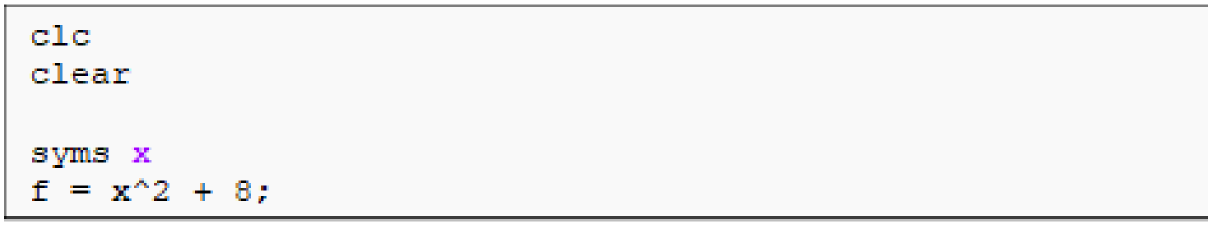
(a) `diff(7*exp(4*x),x,2)`

(b) `diff(7*exp(4*x),x,1)`

(c) `diff(7exp(4*x),x,1)`

(d) `diff(7*exp^(4*x),x,2)`

(3). To find $f(3)$, given $f(x) = x^2 + 8$, the correct line of code to add to the following program is



```
clc
clear

syms x
f = x^2 + 8;
```

(a) `subs(f,3,x)`

(b) `subs(f,x,3)`

(c) `f(3)`

(d) `function(f,3)`

(4). The `diff()` function is used to

(a) find the derivative of a symbolic function.

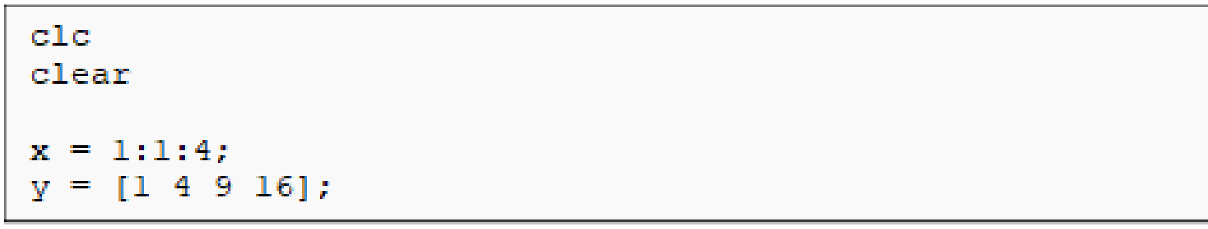
(b) calculate the difference between adjacent elements in a vector.

(c) perform the subtraction of two variables.

(d) both A and B

(5). Given the discrete function, *y(x)*, as the vectors x and y of discrete data points, which of the following would not appear in a program that approximates the first derivative between data pairs?

The following partial program is provided for reference.



```
clc
clear

x = 1:1:4;
y = [1 4 9 16];
```

(a) `diff(y,x,1)`

(b) `diff(x)`

(c) `diff(y)`

(d) `diff(y)./diff(x)`

Problem Set

(1). Use the `diff()` function to find

$\dfrac{d}{dx}(7\sin(4x))$ at $x = 3.5$

(2). Use the `diff()` function to find

$\frac{d^2}{dx^2}\left(4\sin(4x^2) + e^x\right)$ at $x = 3.75$

(3). A rectangle is inscribed in a semi-circle of radius 2m (as shown in Figure A). If the area of the rectangle is to be maximized, find the dimensions of the rectangle.

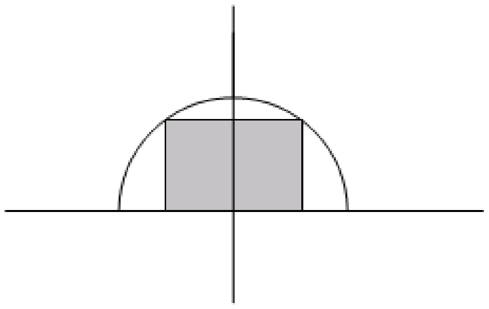


Figure A: Rectangle inscribed inside circle used for Exercise 3.

(4). The velocity of a rocket is given by

$$\nu(t) = 2000\ln\left[\frac{14\times10^4}{14\times10^4 - 2100t}\right] - 9.8t,\ 0 \le t \le 30$$

where ν is given in m/s and t is given in seconds. At $t = 16s$, find the velocity and acceleration of the rocket. Display your results in the command window using the appropriate method.

(5). An experimental single-piston pump is put through a series of tests in a laboratory. The velocity (as a function of time) of the piston in the pump is measured as various forces are applied to the piston. To determine expected bearing life, rod stress, and pumping capacities, the piston force function (as a function of time) must be found. To help determine the efficiency of the pump, the kinetic energy function of the piston must also be determined (assume the piston has no potential or rotational stored energy).

The piston velocity function is

$$v(t) = (3.21)\sin(2.3t) + \cos(2.3t)\ln(5.2t)\ , \qquad 0 < t \leq 10$$

where,

v is the velocity given in meters/second

t is the time given in seconds

Given is the mass of the piston as 0.73 kg. Using MATLAB, write a program that outputs:

(a) the piston force function,$F(t)$,

(b) the piston kinetic energy function, $KE(t)$,

(c) the maximum piston speed, m/s,

(d) the minimum piston speed, m/s.

Hint:

$$\text{Force} = \text{mass} \times \text{acceleration}$$

$$\text{Kinetic Energy} = \frac{1}{2} \times \text{mass} \times \text{velocity}^2$$

(6). There is strong evidence that the first level of processing of what we see is done in the retina. It involves detecting something called edges or positions of transitions from dark to bright or bright to dark points in images. These points usually coincide with boundaries of objects. To model the edges, derivatives of functions such as

$$f(x) = \begin{cases} 1 - e^{-ax}, & x \geq 0 \\ e^{ax} - 1, & x \leq 0 \end{cases}$$

need to be found. Find $f'(0.1)$ and $f'(-0.1)$. Assume $a = 0.24$.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.5: Integral Calculus

Learning Objectives

After reading this lesson, you should be able to:

1) *integrate a continuous function in MATLAB,*

2) *integrate a discrete function in MATLAB.*

What is integration?

Integration is defined as the area under a curve (see Figure 1).

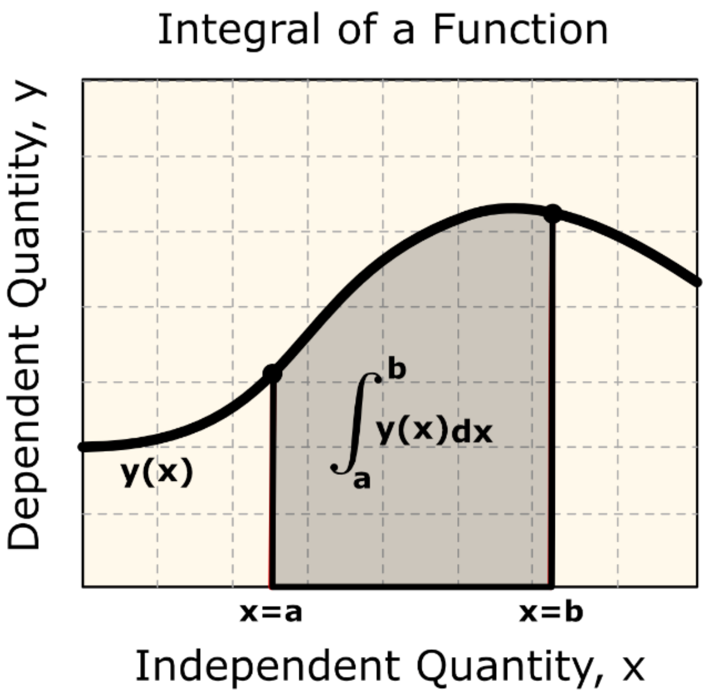


Figure 1: The integral of the function $y(x)$ from the limits of a to b is shown in the shaded area under the curve.

Why would we want to integrate a function? Among the most common examples are finding the velocity of a body from an acceleration function and displacement of a body from a velocity function. Throughout many engineering fields, there are countless applications for integral calculus. Sometimes, these integrals cannot be found exactly.

The general form of an integral is given by

$$I = \int_a^b f(x)dx$$

where,

$f(x)$ is called the integrand,

$a =$ lower limit of integration,

$b =$ upper limit of integration

As you can see from Equation (1), we need four inputs to conduct integration.

1. The function $f(x)$ that needs to be integrated.

2. The variable with respect to which the function needs to be integrated x.
3. The lower limit of integration a.
4. The upper limit of integration b.

The type of integral that requires all four of these inputs is called a definite integral. In contrast, the indefinite integral only requires the first two inputs: the function $f(x)$ and the variable x. MATLAB can conduct both types of integration.

How does MATLAB conduct symbolic integration?

While using MATLAB, several integration functions are available to the programmer. Depending on the type of integration required, the two main functions are the `int()` and `trapz()`. The int() function is used for integrating continuous functions (both indefinite and definite), whereas the `trapz()` function is used when integrating a discrete function. Both of these functions will be discussed in this lesson.

Recall from your integral calculus class that when conducting indefinite integration, a constant (usually "C") must be added to the solution. Look at Example 1 to see the `int()` function in use.

Important Note: MATLAB does not automatically add a constant of integration for an indefinite integral.

Example 1

Use the `int()` function to evaluate the integrals in parts (a) and (b). Evaluate both integrals in the same m-file, and use the `fprintf()` or `disp()` function to display the outputs in the Command Window.

a) $\int_{2.0}^{8.7} e^x \sin(3x) dx$

b) $\int e^x \sin(3x) dx$

Solution

We will use the `int()` function to evaluate both integrals. For part (a) there will be four inputs to the function, and in part (b) we will need two inputs. Notice that the `syms` command must be used before the `int()` function in the m-file to establish the symbolic variables in the integrand. Because the integral in part (b) is indefinite, the "+ C" was added to output the most appropriate solution.

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To integrate a symbolic function
fprintf('To integrate a symbolic function.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
syms x                 %Defining the symbolic variable 'x'
y = exp(x)*sin(3*x);   %Defining the function to integrate
a = 2.0;               %Storing lower limit
b = 8.7;               %Storing upper limit

fprintf('The function to integrate is %s.\n',char(y))
fprintf('The limits of integration are from a=%g to b=%g.\n\n',a,b)

%----------------------------- SOLUTION ----------------------------
intA = int(y,x,a,b);   %Definite integral for part A
intB = int(y,x);       %Indefinite integral for part B

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The value of the definite integral is %g.\n',double(intA))
fprintf('The indefinite integral is %s + C.\n',char(intB))
```

```
Command Window Output                                        Example 1
PURPOSE
To integrate a symbolic function.

INPUTS
The function to integrate is sin(3*x)*exp(x).
The limits of integration are from a=2 to b=8.7.

OUTPUTS
The value of the definite integral is -525.527.
The indefinite integral is -(exp(x)*(3*cos(3*x) - sin(3*x)))/10 + C.
```

Can MATLAB do numerical integration of discrete functions?

As mentioned previously, the `trapz()` function is used when the integrand is given as discrete data points. An example of when to use the `trapz()` function is to integrate a function y given at discrete x-values.

The `trapz()` function uses numerical integration to find the area under a given curve. It joins consecutive data points via straight lines. Using this line and the horizontal axis, a trapezoidal segment (shape) is formed. To do this, the two vectors x and y to be evaluated must be the same size. For example, 10 data pairs would create 9 trapezoids. The summation of the area of these trapezoids is estimated as the value of the integral. This math behind the numerical method is not required to use `trapz()`, as the inputs are simply two vectors, but it is important to understand for any real-world application of the method. Example 2 shows the use of the `trapz()` function.

✓ ***Important Note:*** `trapz(x,y)` approximates the integral from `min(x)` to `max(x)`.

Example 2

Input the x and y data into a new MATLAB m-file.

Table 1: Data pairs to be used for Example 2.

x	y
0	2.01
1	3.97
3	20.2
5	50.95
6	72.76
9	166.5

Plot y vs. x on a standard linear plot where y is the vertical axis and x is the horizontal axis. Use the `trapz()` function to find the area under the curve from $x = 0$ to $x = 9$. Output the resulting value using the `fprintf()` function to the Command Window.

Solution

Because the data provided is a discrete function, the `trapz()` function will be used for the integration. Remember, in order for the `trapz()` function to work, the two data arrays must be the same size. In this case, the two vectors both have 6 elements.

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To integrate a discrete function
fprintf('To integrate a discrete function.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
x = [0 1 3 5 6 9];                         %Inputting x vector
y = [2.01 3.97 20.2 50.95 72.76 166.5];  %Inputting y vector
disp('Independent variable, x =')
disp(x)
disp('Dependent variable, y =')
disp(y)

%------------------------------ SOLUTION ----------------------------
trapzInt = trapz(x,y);  %Approximating the area under the curve

%Plotting the discrete function to visualize
figure                          %Creating new figure
plot(x,y,'o')                   %Plotting points as markers
hold on                         %Telling MATLAB to plot on the same plot
grid on                         %Placing grid on plot
plot(x,y)                       %Plotting points as lines
hold off
%Figure formatting
title('Data Pairs (x,y)')
xlabel('Independent variable, x')
ylabel('Dependent variable, y')

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The area under the curve from x=%g to x=%g is %g\n',...
           min(x),max(x),double(trapzInt))
```

```
Command Window Output                                    Example 2

PURPOSE
To integrate a discrete function.

INPUTS
Independent variable, x =
     0     1     3     5     6     9

Dependent variable, y =
    2.0100    3.9700   20.2000   50.9500   72.7600  166.5000

OUTPUTS
The area under the curve from x=0 to x=9 is 519.055
```

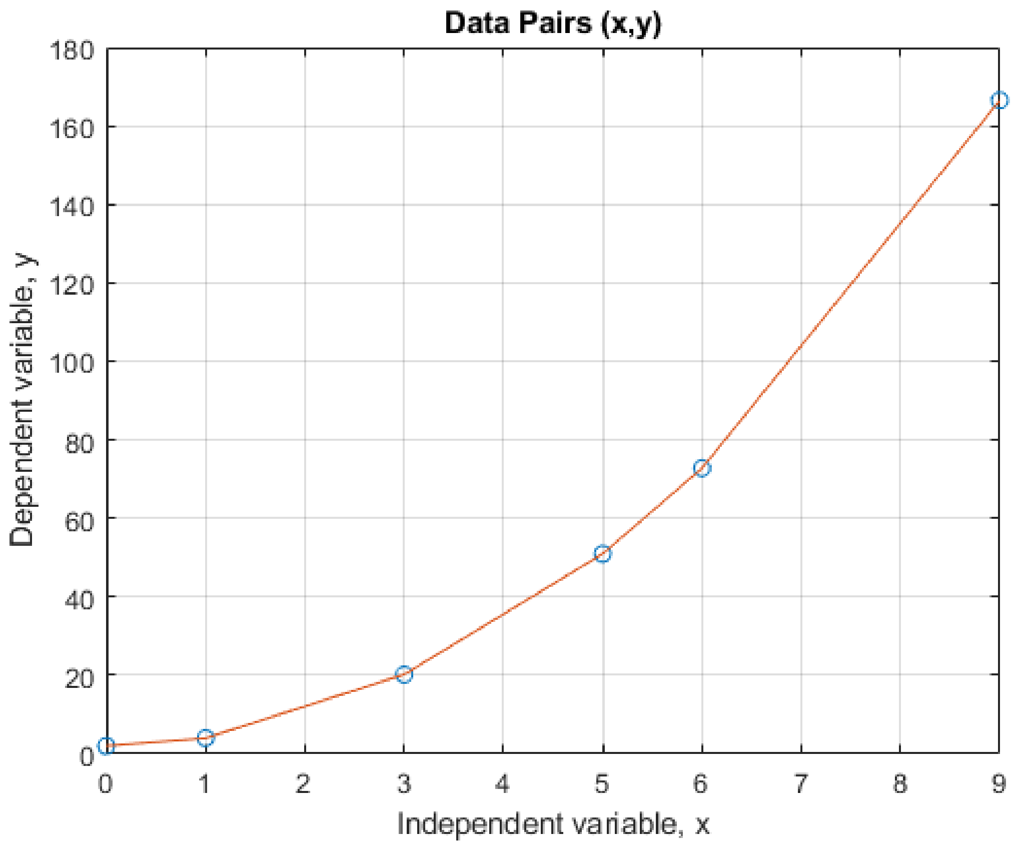


Figure 2: Figure of plotted points for Example 2.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Definite integral of a symbolic function	`int()`	`int(y,x,a,b)`
Indefinite integral of a symbolic function	`int()`	`int(y,x)`
Approximate the area under a curve given by discrete data points	`trapz()`	`trapz(x,y)`

Multiple Choice Quiz

(1). The MATLAB function for integration of a symbolic function is

(a) `integrate()`

(b) `diff()`

(c) `integ()`

(d) `int()`

(2). The output of `trapz()` is

(a) a mathematical function that is best fit to the given data.

(b) a single number that is the approximate area under the curve.

(c) a symbolic function that is the integral of the input discrete function.

(d) a vector of values.

(3). To find $\int_4^8 7e^{-4x}dx$, the correct line of code to add to the following program is

(a) `int(7\*exp(-4\*x),x,4,8)`

(b) `int(7\*exp(-4\*x),x,8,4)`

(c) `int(7exp\^(-4\*x),x,4,8)`

(d) `int(7exp(-4x),x,4,8)`

(4). To find$\int_{12}^8 (x^2+2)dx$, the correct line of code to add to the following program is

(a) `int(x\^2+2,x,8,12)`

(b) `int(x\*\*2,x,8,12)`

(c) `int(x\^2+2,x,12,8)`

(d) `int(x\^2+2,x,8,12)`

(5). To numerically integrate a discrete function, y calculated as a function of x from $x = 1$ to $x = 10$ using trapezoidal segments, the correct line of code to add to the following program is

(a) `trapz(x,y,1,10)`

(b) `trapz(y,x,1,10)`

(c) `trapz(x,y)`

(d) `trapz(y,x)`

Problem Set

Note: In these exercises, if a vector is raised to a power, remember to use the (.) operator. For example, if you want to square each element of a vector `vec`, you will do this by using `vec.^2`. Similarly, if you want to find the reciprocal of each element of the vector `vec`, you will do this by using `1/.vec`.We took similar measures before when plotting in Module 3. The full explanation for this procedure is contained in the next lesson (Lesson 4.6).

(1). Use the `int()` function to find $\int_2^8 e^{4x}\cos(x)dx$

(2). Use the `int()` function to find$\int_{3.4}^{8.2} f(x)dx$

where

$$f(x) = \begin{cases} \&x^2, & 0 \le x < 4, \\ \&x^3, & 4 \le x < 10. \end{cases}$$

(3). Find the vertical distance covered by a rocket from $t = 8$ to $t = 30$ seconds by solving the integral below.

$$x = \int_8^{30} \left(2000 \ln\left[\frac{140000}{140000 - 2100t}\right] - 9.8t\right) dt$$

(4). A company advertises that every roll of toilet paper has at least 250 sheets. The probability that there are 250 or more sheets in the toilet paper is given by

$$P(y \geq 250) = \int_{250}^{\infty} 0.3515 e^{-0.3881 \ (y-252.2) \ ^2} dy.$$

Find the value of the integral (thus finding the probability) using the `int()` function.

(5). Use the `trapz()` function to find $\int_{1.2}^{8.6} z dz$, given

$$z = \ln\left(r^2\right) r$$

(6). A trunnion of diameter $12.363''$ has to be cooled from a room temperature of $80^{\circ}F$ before it is shrink-fitted into a steel hub (Figure A).

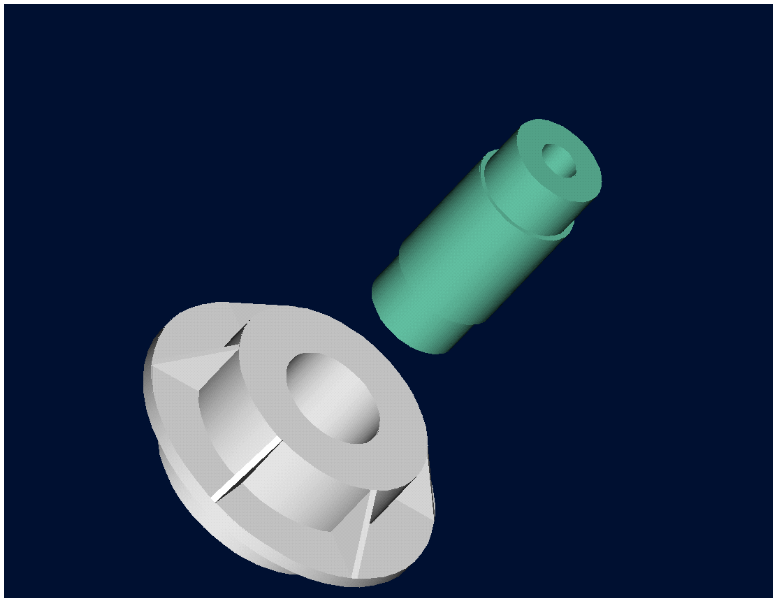


Figure A: The hub and trunnion prior to shrink fitting.

The equation that gives the diametric contraction (in inches) of the trunnion by immersing it in a dry-ice/alcohol (temperature is $-108^{\circ}F$) mixture is given by

$$\Delta D = 12.363 \int_{80}^{-108} \left(-1.2278 \times 10^{-11} T^2 + 6.1946 \times 10^{-9} T + 6.015 \times 10^{-6}\right) dT$$

Use `int()` function to find the diametric contraction of the trunnion and the new trunnion diameter. Output the program results to the Command Window with a brief description.

(7). Find $\int_{3.1}^{8.5}(x^3 - 4x^2 + 7x)dx$ using both the `trapz()` and `int()` functions. Compare the outputs in the Command Window using `fprintf()` function. Show at least 6 decimal places for all outputs.

(8). Given the following $(x,\ y)$ data

(3, 2), (5.4, 4.6), (6.2, 5.1), (7.9, 6.7), (11.0, 8.5), (15.7, 14.0)

Use `trapz()` function to find $\int_{3}^{15.7} y dx$.

(9). The error function is defined as $erf(y) = \frac{2}{\sqrt{\pi}}\int_0^y e^{-z^2}dz$.

Using a single m-file find the values of *erf*(0.2), *erf* (0.5), and *erf* (3.4). Compare these three results to the MATLAB output for the error function `erf()`. Use the `fprintf()` function to compare and output the results.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.6 – Linear Algebra

Learning Objectives

After reading this lesson, you should be able to:

1) *perform linear algebra operations on matrices,*
2) *solve systems of equations with MATLAB.*

Note that the basics of matrices and vectors in MATLAB were covered in Lesson 2.6. We do not repeat them here.

What is linear algebra?

Linear algebra is the set of rules and operations dealing with equations that contain matrices (for the scope of this book). In this lesson, we review the basics of linear algebra: how to add, subtract, and multiply matrices and vectors and a few other common operations. However, our main focus will be on MATLAB. It is essential to learn these basics since vectors and matrices are very common in programming for a variety of applications. We have included a primer on linear algebra in Appendix A in case you need further practice with the fundamentals of linear algebra. If you do not know how to do the operations discussed in this lesson on paper, be sure to get up to speed on them. Doing so will save you time in the long run as programming them requires a fundamental understanding in most cases.

✓ ***Important Note:*** Remember a vector is a special type of matrix: a single column or single row matrix. Therefore, everything said about matrices in this lesson includes vectors as well.

How do I add and subtract matrices?

To add or subtract two matrices, the two matrices have to be the same dimension (both matrices need to have the same number of rows and columns). Figure 1 shows an example of matrix addition, where two rectangular matrices of size 3×2 are added together. Example 1 demonstrates how to perform matrix addition in MATLAB.

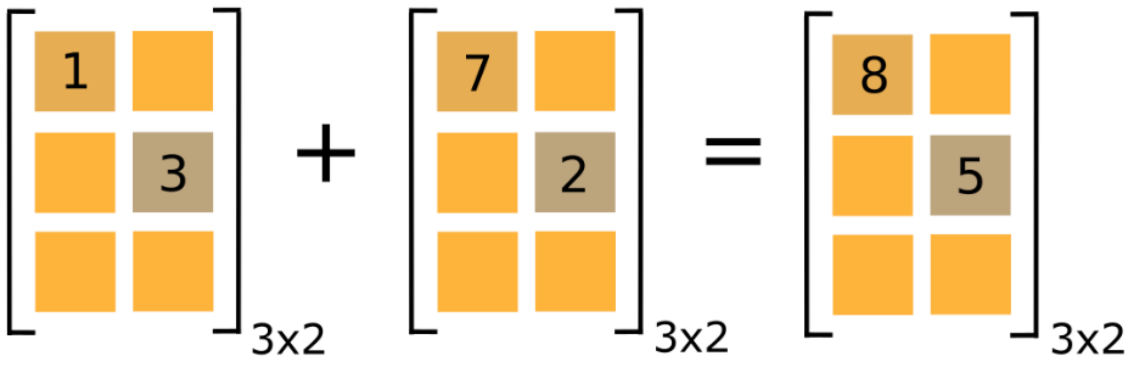


Figure 1: Example of matrix addition. Only the addition of the two elements is shown in this visual example.

Can I use math functions like `sin()` on matrices?

As mentioned in previous lessons, to find the *sine* of each element in a matrix, just give MATLAB the matrix as an input. That is, if $[A]$ is your matrix, `sin(A)` will return a matrix of the sine of each element in $[A]$. The same is true for all the trigonometric, exponential, and logarithmic functions. As you can see in Example 1, the same mathematical functions (`sin()`, `log()`, `exp()`, etc.) can be used on matrices with no change in syntax.

Example 1

Input the following two matrices, $[A]$ and $[B]$, into a new m-file.

$$[A] = \begin{bmatrix} 1 & 2 & 6 \\ 2 & 65 & 1 \\ 4 & 2 & 1 \end{bmatrix} \; [B] = \begin{bmatrix} 2 & 3 & 4 \\ 2 & 1 & 45 \\ 9 & 3 & 6 \end{bmatrix} \; [\mathrm{C}] = \begin{bmatrix} 0 & \pi \\ \frac{\pi}{2} & 1 \end{bmatrix}$$

Conduct the following operations:

a) Add the two matrices $[A]$ and $[B]$

b) Find the cosine of C(1,1)

c) Find the cosine of all the elements in $[C]$

Solution

MATLAB Code example1.m

```
clc
clear

%---------------------------- PURPOSE ----------------------------
fprintf('PURPOSE\n')
%To conduct mathematical operations on matrices
fprintf('To conduct mathematical operations on matrices.\n\n')
```

MATLAB Code (continued) example1.m

```matlab
%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining matrices
A = [1 2 6;2 65 1;4 2 1];
B = [2 3 4;2 1 45;9 3 6];
C = [0 pi; pi/2 1];

%Displaying matrices
disp('Matrix [A]:')
disp(A)
disp('Matrix [B]:')
disp(B)
disp('Matrix [C]:')
disp(C)

%----------------------------- SOLUTION ----------------------------
%Part a)
D    = A + B;         %Adding matrices A and B
%Part b)
C11  = cos(C(1,1));   %Taking the cosine of C(1,1)
%Part c)
Ccos = cos(C);        %Taking the cosine of the entire matrix C

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
%Displaying part a)
disp('Part a) A + B =')
disp(D)
%Displaying part b)
fprintf('Part b) The cosine of the first element of matrix C: %g\n', C11)
%Displaying part c)
fprintf('Part c) Cosine of matrix C\n')
disp(Ccos)
```

Command Window Output Example 1

```
PURPOSE
To conduct mathematical operations on matrices.

INPUTS
Matrix [A]:
     1     2     6
     2    65     1
     4     2     1

Matrix [B]:
     2     3     4
     2     1    45
     9     3     6

Matrix [C]:
         0    3.1416
    1.5708    1.0000
```

Command Window Output (continued) Example 1

```
OUTPUTS
Part a) A + B =
     3     5    10
     4    66    46
    13     5     7

Part b) The cosine of the first element of matrix C: 1
Part c) Cosine of matrix C
    1.0000   -1.0000
    0.0000    0.5403
```

How do I perform matrix multiplication?

Unlike addition and subtraction, matrix multiplication is not simply the product of each set of elements: e.g., A(1,1)*B(1,1). You do not absolutely need to understand how to do this by hand, but you must understand two resulting facts:

1. Inner dimensions of a matrix product must be equal: $\mathrm{A}_{\mathrm{m}\times\mathrm{n}} \times \mathrm{B}_{\mathrm{n}\times\mathrm{p}}$. That is, the number of columns of the first matrix has to be equal to the number of rows of the second matrix.
2. Matrix products are not commutative. That is, in general, the order of the two matrices matters ([A]*[B] does not equal [B]*[A]).

An example of matrix multiplication can be seen in Figure 3, while a full MATLAB code of multiplying two matrices together is shown in Example 3.

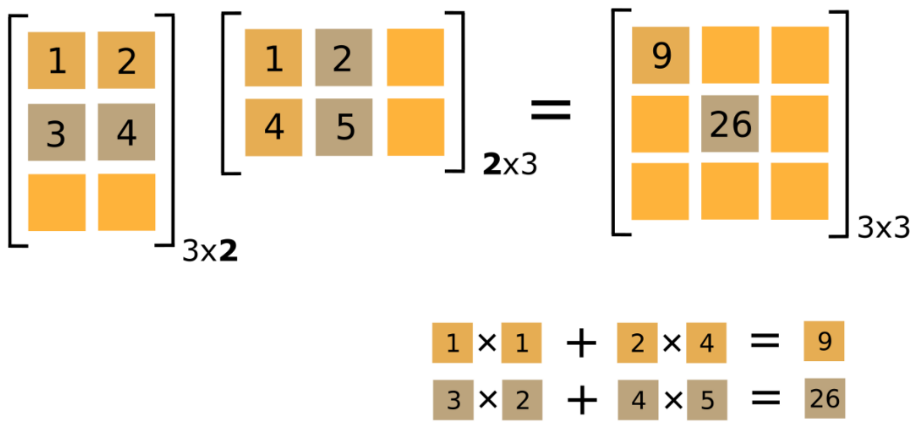


Figure 3: An example of matrix multiplication. Only two calculated elements are shown.

What is the difference between matrix and array operations?

Array operations performed element-by-element operations to matrices (and vectors). Matrix operations are traditional matrix algebra operations as outlined in the previous sections of this lesson and in Appendix A. Array operations have a dot next to the math operator. For example, `A*B` would perform a matrix multiplication of matrix `A` and `B`. However, matrix `A.*B` would multiply each element of `A` and `B` together (not the same answer as you can see in Example 2!), and requires the matrices to have equal dimensions.

Example 2

Input the following two matrices, $[A]$ and $[B]$, into a new m-file.

$$[A] = \begin{bmatrix} 1 & 2 & 6 \\ 2 & 65 & 1 \\ 4 & 2 & 1 \end{bmatrix} \quad [B] = \begin{bmatrix} 2 & 3 & 4 \\ 2 & 1 & 45 \\ 9 & 3 & 6 \end{bmatrix}$$

Conduct the following operations:

a) Multiply matrix $[A]$ by $[B]$.

b) Conduct element to element multiplication of the two matrices $[A]$ and $[B]$.

Solution

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To conduct mathematical operations on matrices
fprintf('To conduct mathematical operations on matrices.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining matrices
A = [1 2 6;2 65 1;4 2 1];
B = [2 3 4;2 1 45;9 3 6];
%Displaying matrices
disp('Matrix [A]:')
disp(A)
disp('Matrix [B]:')
disp(B)

%----------------------------- SOLUTION ----------------------------
%Part a) Multiply A*B
D = A*B;

% Part b) Find: A.*B
E = A.*B;  %Performing element-by-element multiplication of A times B
             %   and multiplying the product by a scalar value of 2
```

MATLAB Code (continued) example2.m

```
%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
disp('Part a) A * B =')
disp(D)
disp('Part b) A.*B =')
disp(E)
```

Command Window Output Example 2

```
PURPOSE
To conduct mathematical operations on matrices.

INPUTS
Matrix [A]:
     1     2     6
     2    65     1
     4     2     1

Matrix [B]:
     2     3     4
     2     1    45
     9     3     6

OUTPUTS
Part a) A * B =
          60          23         130
         143          74        2939
          21          17         112

Part b) A.*B =
     2     6    24
     4    65    45
    36     6     6
```

As you can see, as long as the rules for matrix operations are followed (as given in Appendix A), adding, subtracting and multiplying matrices requires no different symbols than from what we are used to for simple scalar operations. However, recall that matrix division is undefined. To solve this problem, the MATLAB programmers introduced a function that finds the inverse of a square matrix.

Table 1: Several commonly used matrix functions and operators. The inputs listed are the matrices $[A]$ and $[B]$ (where applicable).

Operation	**Syntax**	**Function**	**Rules**
Addition	`A+B`	Adds two matrices.	All dimensions must be equal.
Subtraction	`A-B`	Subtracts two matrices.	All dimensions must be equal.
Multiplication (matrix)	`A*B`	Multiplies two matrices.	Inner dimensions must be equal.
Multiplication (by scalar)	`5*B`	A scalar times a matrix.	Multiplies each element of the matrix by the scalar. No array operator needed.

Operation	**Syntax**	**Function**	**Rules**
Array operations	`.`	Conduct a specified element to element operation (see usage example below).	Depends on usage.
Multiplication (array)	`A.*B`	Multiplies two matrices element-by-element.	Matrices must have equal dimensions. Array operator required.
Matrix inverse	`inv(A)`	Outputs the inverse of a matrix.	Must be a square matrix.
Exponent (matrix)	`A^2`	Equivalent to $[A]$*$[A]$	Must be a square matrix.
Exponent (array)	`A.^2`	Raise each element, A(i,j), to power	Can be any size.

Notice that in Table 1, there is no function to conduct matrix division. This is because matrix division is not defined for matrices. However, one may also use the arithmetic division operator with the array (.) operator to conduct element-by-element division of one matrix by another.

How do I take the inverse of a matrix?

Recall that multiplying a single number (scalar) by its inverse will give a product equal to one (e.g., $5 \times 5^{-1} = 1$). Similarly, for matrices, multiplying a matrix by its inverse will yield the identity matrix: all diagonal matrix elements are equal to "one" and all non-diagonal elements are zero. An example of an identity matrix is shown in Figure 3. In MATLAB, the inverse of a square matrix can be found using the function `inv()`. The inverse of a matrix is a key concept in linear algebra, so it is important to understand it on a conceptual basis even for programming.

Example 3

Input the following two matrices, $[A]$ and $[B]$ into an m-file.

$$[A] = \begin{bmatrix} 1 & 2 & 6 \\ 2 & 65 & 1 \\ 4 & 2 & 1 \end{bmatrix} \; [B] = \begin{bmatrix} 2 & 3 & 4 \\ 2 & 1 & 45 \\ 9 & 3 & 6 \end{bmatrix}$$

Conduct the following operations:

a) Find the inverse of matrix $[A]$, naming it **`invA`**. Now, multiply matrix $[A]$ by **`invA`** and examine the resulting matrix.

b) Find the size of matrix $[A]$ and develop an identity matrix of this size, naming it ‘ID. Multiply matrix $[A]$ by `ID` and examine the resulting matrix.

Solution

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To explore the inverse of a matrix and identity matrices
fprintf('To explore the inverse of a matrix and identity matrices.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
A = [1 2 6;2 65 1;4 2 1];  %Defining matrix
B = [2 3 4;2 1 45;9 3 6];  %Defining matrix

%Displaying matrices
disp('Matrix [A]:')
disp(A)
disp('Matrix [B]:')
disp(B)

%------------------------------ SOLUTION ----------------------------
% Part a) find: A*inv(A)
invA = inv(A);             %Finding the inverse of A
R    = A*invA;             %Calculating inv(A)*A
%Part b) Find: A*I
[rowsA,colsA] = size(A);   %Finding the size of A
ident = eye(rowsA,colsA);  %Creating an identity matrix the same size as A
E      = A*ident;

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('Part a) A*inv(A) =\n')
disp(R)
fprintf('This is an identity matrix!\n\n')

fprintf('Part b) A*ident\n')
disp(E)
fprintf('This matrix is the same as matrix A!\n')
```

Examining the output of part (a) shows an identity matrix - this is what should be expected. In part (b) of Example 3, an identity matrix is multiplied to matrix

[A], and the resulting matrix is the same as matrix [A]. This is also in line with our expectations.

Command Window Output Example 3

```
PURPOSE
To explore the inverse of a matrix and identity matrices.
```

Command Window Output (continued) Example 3

```
INPUTS
Matrix [A]:
     1     2     6
     2    65     1
     4     2     1

Matrix [B]:
     2     3     4
     2     1    45
     9     3     6

OUTPUTS
Part a) A*inv(A) =
    1.0000         0         0
         0    1.0000    0.0000
         0    0.0000    1.0000

This is an identity matrix!

Part b) A*ident
     1     2     6
     2    65     1
     4     2     1

This matrix is the same as matrix A!
```

Can MATLAB do advanced matrix and vector operations?

There are a significant number of matrix operations available in MATLAB as predefined functions, and to cover them all would be out of the scope of this book. A few more common functions are shown in Table 2, and their usage is shown in Example 4. If you do not see the matrix operation you are interested in, try searching for the operation in MATLAB Help.

In Table 2, the functions to conduct a vector dot/cross product can be modified to work with matrices. To do this operation, an additional input to the function is required (adding the dimension after the listed inputs). Use the MATLAB help menu and search the function name for more details (e.g., `>>doc max`).

The `norm()` function has several norm types to choose from. For example, Type 1 (i.e., `norm(A,1)`) is the largest column sum: $\|A\|_1 = \max_{\mathrm{j}} \left(\sum_{i=1}^{m} A_{ij} \right), j = 1, ..., n$ of an $m \times n$ matrix.

The `sort()` function can also be modified from what is shown in Table 2 to numerically sort a vector from the largest number to the smallest by using the inputs `sort(A,'descend')`. If the `sort()` function is used as shown in Table 2, it will default to sort a matrix from the smallest element to the largest element (in ascending order).

Table 2: Commonly used matrix functions. The input(s) to the listed functions is/are a matrix or vector[A] and [B] (where applicable).

Task	**Syntax**	**Explanation**
Vector dot product	`dot(A,B)`	Outputs the dot product of two vectors of equal length.
Vector cross product	`cross(A,B)`	Outputs the cross product of two vectors of equal length.
Matrix transpose	`A'`	Finds the transpose of a matrix.
Matrix Transpose	`transpose(A)`	Finds the transpose of a matrix.
Smallest element	`min(A)`	Outputs the smallest element in a vector.
Largest element	`max(A)`	Outputs the largest element in a vector.
Sort an array	`sort(A)`	Sorts a vector from least to greatest.
Matrix determinant	`det(A)`	Outputs the scalar determinant of a square matrix.
Norm of a matrix	`norm(A,p)`	Finds the norm of a matrix.
Trace of a matrix	`trace(A)`	Outputs the scalar value of the trace of a matrix

Example 4

Input the following two vectors, [A] and [B], and the matrix [C] into a new m-file.

$$[A] = \begin{bmatrix} 2 & 4 & -2 \end{bmatrix} [B] = \begin{bmatrix} 0 & 7 & 1 \end{bmatrix} [C] = \begin{bmatrix} 0 & -3 & 0 \\ 2 & 5 & 6 \\ 1 & 0 & -13 \end{bmatrix}$$

Using the functions listed in Table 2, complete the following:

a) Find the vector cross product of vectors $[A]$ and $[B]$, and name this new vector `crossAB`.

b) Find the value of the maximum and minimum element of vector `crossAB` from part (a).

c) Find the numeric values of the trace, norm (maximum column sum), and determinant of matrix $[C]$.

Solution

MATLAB Code example4.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To conduct more advanced matrix operations
fprintf('To conduct more advanced matrix operations.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
A = [2 4 -2];
B = [0 7  1];
C = [0 -3 0;2 5 6;1 0 -13];
disp('Vector [A]:')
disp(A)
disp('Vector [B]:')
disp(B)
disp('Matrix [C]:')
disp(C)

%----------------------------- SOLUTION ----------------------------
%Part a)
crossAB = cross(A,B);  %Finding the cross product of A cross B
%Part b)
minVC = min(crossAB);  %Finding the min of the vector crossAB
maxVC = max(crossAB);  %Finding the max of the vector crossAB

%Part c)
traceC = trace(C);     %Finding the trace of matrix C
normC  = norm(C,1);    %Finding the norm (type 1) of matrix C
detC   = det(C);       %Finding the determinant of matrix C

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('Part a)\n')
fprintf('  The cross product of A and B is:\n')
disp(crossAB)
fprintf('Part b)\n')
fprintf('  The maximum value is %g.\n',maxVC)
fprintf('  The minimum value is %g.\n\n',minVC)
fprintf('Part c)\n')
fprintf('  Trace of [C] is %g.\n',traceC)
fprintf('  The 1-norm of [C] is %g.\n',normC)
fprintf('  Determinant of [C] is %g.\n',detC)
```

```
Command Window Output                                    Example 4
PURPOSE
To conduct more advanced matrix operations.
```

```
Command Window Output (continued)                        Example 4
INPUTS
Vector [A]:
     2     4    -2

Vector [B]:
     0     7     1

Matrix [C]:
     0    -3     0
     2     5     6
     1     0   -13

OUTPUTS
Part a)
  The cross product of A and B is:
    18    -2    14

Part b)
  The maximum value is 18.
  The minimum value is -2.

Part c)
  Trace of [C] is -8.
  The 1-norm of [C] is 19.
  Determinant of [C] is -96.
```

How can I solve systems of equations with MATLAB?

One of the ways we can use MATLAB to harness the power of the computer is by it solving large systems of equations for us. A set of three equations and three unknowns may not be terribly hard to solve by hand, but there are many applications where that number is in the thousands.

To cut down on algebraic manipulation, matrices are commonly used to solve linear systems of equations. As outlined in Appendix A, to solve any linear system of equations a programmer must have (1) the coefficient matrix A, (2) solution or unknown vector x, and (3) the right-hand side vector C. These three vectors/matrices are related in the following form $[A] \cdot [x] = [C]$. Examples 5 and 6 demonstrate this process in MATLAB.

Example 5

A coworker is having trouble solving a system of 3 equations with 3 unknowns and has asked for your help. You tell them "I'll have the system of equations solved for you before lunch!"

The equations are

$$z = 2.1x + 43.03y - 54.3z = -201.0y + 4.1z = -x - 21.65y - 43.1$$

Solution

Rewrite the equations so that the unknown variables are on the left side and the known constants are on the right side:

$$2.1x + 43.03y - z = 54.3 - 201.0y - z = -4.1 - x - 21.65y - z = 43.1$$

The set of equations can be written in matrix form as

$$\begin{bmatrix} 2.1 & 43.03 & -1 \\ 0 & -201.0 & -1 \\ -1 & -21.65 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -54.3 \\ -4.1 \\ 43.1 \end{bmatrix}$$

This system of equations has been annotated for clarity in Figure 7. Each of the three arrays (coefficient matrix, solution vector, right-hand side vector) are labeled.

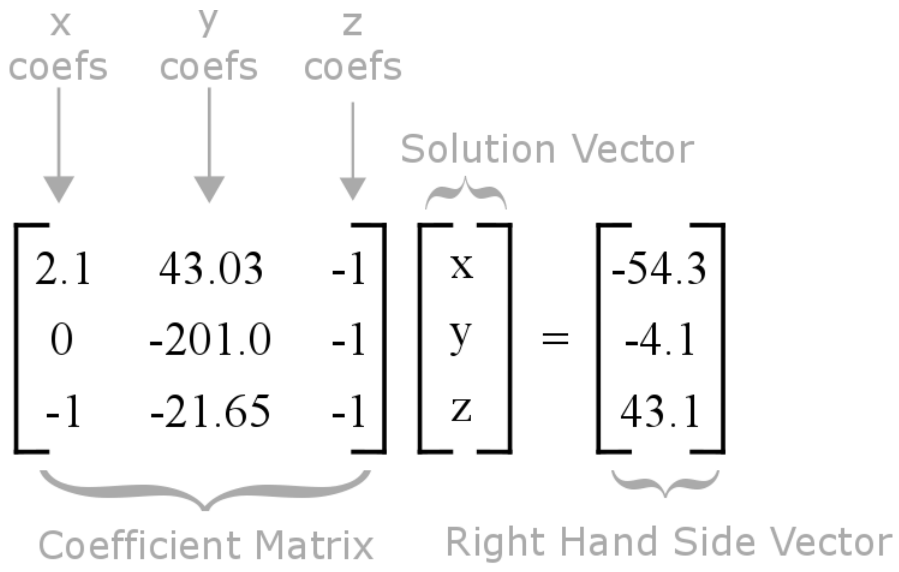


Figure 7: Annotated system of equations for Example 5.

Now that the coefficient matrix, the right-hand side, and solution vectors are found, one can write a program to solve the system of equations. Functions listed previously in this lesson, such as the inverse function, `inv()`, are used to help solve this system of equations.

MATLAB Code example5.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To solve a system of equations
fprintf('To solve a system of equations.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
coef = [2.1 43.03 -1;
    0 -201.0 -1;
    -1 -21.65 -1];       %Defining the coefficient matrix

rhs = [54.3;-4.1;43.1];  %Defining the right hand side vector
fprintf('The coefficient matrix is:\n')
disp(coef)
fprintf('The right-hand side vector is:\n')
disp(rhs)

%------------------------------ SOLUTION -----------------------------
%The equation for a system of equations is A*x = C
%   Equivalently, we use [coef]*[unknown] = [rhs]
unknown = inv(coef)*rhs;

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
%Assigning each element of unknown to its corresponding variable (optional)
x = unknown(1);
y = unknown(2);
z = unknown(3);
fprintf('The solution to the set of linear equations is:\n')
fprintf('    x = %g, y = %g, z = %g\n',x,y,z)
```

```
Command Window Output                                        Example 5
PURPOSE
To solve a system of equations.

INPUTS
The coefficient matrix is:
    2.1000   43.0300   -1.0000
         0 -201.0000   -1.0000
   -1.0000  -21.6500   -1.0000

The right-hand side vector is:
   54.3000
   -4.1000
   43.1000

OUTPUTS
The solution to the set of linear equations is:
    x = -1.68235, y = 0.253792, z = -46.9123
```

As mentioned above, matrix division is not defined, which is why we used the `inv()` function in Example 5. This is categorically true. However, MATLAB has implemented a more efficient method for solving systems of equations of the form $[A] \cdot [B] = [C]$, which does not use the inverse of a matrix. This is of the form `x = A\C` or `x = mldivide(A,C)`: these two are equivalent syntax in MATLAB. Note that these are not replacements for calculating the inverse of a matrix.

In Example 6, we solve a system of equations (of the form $[x] \cdot [A] = [C]$) using the `x = A\C` syntax. If you have a system of equations of the form `x*A = C`, you should solve using `x = A/C`. Remember, the order of matrix multiplication is not commutative, so these two cases require different solutions.

Example 6

The amount of force that is exerted on a body can be determined by measuring the amount that the body displaces, and vice-versa. Ultimately, this force value can help determine the amount of stress acting on and inside the body or structure. This method for determining stress is widely used in finite element modeling and is also applicable to a variety of other systems (heat transfer, vibration analysis, etc.). Typically, the force and displacement values are modeled as vectors, and they are related to each other through the stiffness matrix. The relationship of the force, displacement, and stiffness matrices is

$$[\vec{F}] = [k]\ [\vec{x}]$$

where,
F is the force column vector.
k is the stiffness matrix.
x is the displacement column vector.

For each system, the stiffness matrix is different and is dependent on the material properties and the system component geometry in use. Given the stiffness matrix ([k], lbs/in), find the displacement (inches) of all three components if each component has a normal force acting on it as listed (use relationship given above). The element forces are: 9200, 42100, 105630 lbs.

The stiffness matrix is

$$[k] = \begin{bmatrix} 2 & 0.1 & 0.2 \\ 0.1 & 0.002 & 2.1 \\ 0.2 & 2.1 & 0.03 \end{bmatrix} \times 10^3 \text{ kip/in}$$

Output the maximum value of the displacement vector elements with a brief description.

Solution

MATLAB Code example6.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To solve a system of equations using x=A\C
fprintf('To solve a system of equations using x=A\\C.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
F = [9200; 42100; 105630];  %Defining force column vector (lbs)
k = [2 0.1 0.2;             %Defining stiffness matrix (lbs/in)
    0.1 0.002 2.1;
    0.2 2.1 0.03]*10^6;

disp('Forces (lbs):')
disp(F)
disp('Stiffness Matrix (lbs/in):')
disp(k)

%------------------------------ SOLUTION -----------------------------
%The equation for a spring force is [F]=[k][x]
x = k\F;  %Solving for [x]

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The three components displaced (inches):\n')
disp(x)
fprintf('The max displacement is %.4f inches\n',max(x))
```

It is important to note that MATLAB does not account for units, so one must be very careful when solving engineering problems. This example shows a basic way to solve a problem that uses the finite element method (FEM). It should be noted that the element stiffness matrix for most finite element problems is of very large size, but the matrix operations are essentially the same as shown in Example 6.

Command Window Output Example 6

```
PURPOSE
To solve a system of equations using x=A\C.

INPUTS
Forces (lbs):
        9200
       42100
      105630
```

```
Command Window Output (continued)                                    Example 6
Stiffness Matrix (lbs/in):
     2000000      100000      200000
      100000        2000     2100000
      200000     2100000       30000

OUTPUTS
The three components displaced (inches):
    0.0001
    0.0500
    0.0200

The max displacement is 0.0500 inches
```

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Solve A*x = C for x	`mldivide()` or `\`	`mldivide(A,C)` or `A\C`
Solve x*A = C for x	`mrdivide()` or `/`	`mrdivide(C,A)` or `C/A`
Matrix addition	`+`	`A+B`
Matrix subtraction	`-`	`A-B`
Matrix multiplication	`*`	`A*B`
Array operations	`.`	`A.*B`
Matrix inverse	`inv()`	`inv(A)`
Raise a matrix to a power	`^`	`A^2`
Vector dot product	`dot()`	`dot(A,B)`
Vector cross product	`cross()`	`cross(A,B)`
Matrix transpose	`'` or `transpose()`	`A'`
Find the smallest element	`min()`	`min(A)`
Find the largest element	`max()`	`max(A)`
Sort an array	`sort()`	`sort(A)`
Matrix determinant	`det()`	`det(A)`
Norm of a matrix	`norm()`	`norm(A,type)`
Trace of a matrix	`trace()`	`trace(A)`

Multiple Choice Quiz

(1). The array operator (.) in MATLAB is used for

(a) conducting element-by-element matrix operations

(b) conducting matrix multiplication

(c) denoting a decimal point

(d) finding the dot product of vectors

(2). The function to find the determinant of a matrix $[A]$ is

(a) `|A|`

(b) `det(A)`

(c) `determinant(A)`

(d) `determ(A)`

(3). When *subtracting* two matrices, the matrices must have

(a) equal inner dimensions

(b) zeros along the diagonal

(c) unequal sizes

(d) equal sizes

(4). Which of the following will give the transpose of a matrix $[A]$ in MATLAB?

(a) `trans(A)`

(b) `transpose(A)`

(c) `A'`

(d) Both b & c

(5). When *multiplying* two matrices, the matrices must have

(a) equal sizes

(b) zeros along the diagonal

(c) unequal sizes

(d) equal inner dimensions

Problem Set

(1). Given

$$[A] = \begin{bmatrix} 5 & 2 & 4 \\ -2 & 3 & 1 \\ 7 & 2 & -1 \end{bmatrix} \quad [B] = \begin{bmatrix} 4 & -3 \\ 6 & 2 \\ -4 & 1 \end{bmatrix} \quad [C] = \begin{bmatrix} 6 & 9 \\ 5 & -2 \\ 1 & 4 \end{bmatrix}$$

use MATLAB to find

(a) $[B] + [C]$

(b) $[B] - [C]$

(c) inverse of $[A]$

(d) $[A][B]$

(e) $\det(A)$

(f) $[A]^T$

(g) The square of each element of $[A]$

(h) The natural log of each element of $[A]$

(2). Using matrices and MATLAB, solve for the variables x,y,z in the system of linear equations.

$$3x + 5y - 2z = 12x - 2y + 7z = 3x + 8y - 4z = -6$$

Use `fprintf()` to display the solution to the system of equations. Include a brief description of the solution as well.

(3). Given that the equation of a line is $y = mx + b$ where m is the slope and b is the vertical (y-intercept), use matrices to find the equation of the line that passes through the (x,y) data pairs: (1,6) and (4,10).

(4). Given

$$[P] = \begin{bmatrix} 4 & 0 & -3 & 7 \\ 9 & 7 & 4 & 2 \\ 0 & 1 & -9 & 6 \\ 3 & 2 & 7 & 1 \end{bmatrix}$$

use MATLAB to find the

(a) row and column dimensions using the `size()` function.

(b) infinity norm of $[P]$

(c) trace of $[P]$.

(d) inverse of $[P]$, and name the matrix, `Q`.

For parts (a), (b) and (c), use `fprintf()` to display the result.
For (d), use `disp()` to display Q.

(5). Given the two vectors,

$$\vec{\omega} = [4 \quad 2 \quad 7] \vec{r} = [2.5 \quad 4 \quad 1.3]$$

use MATLAB to find the

(a) vector dot product.

(b) vector cross product.

(c) sorted vector of $\vec{r}$, and name the array, *rSort.*

Use the `fprintf()` or `disp()` functions to display the appropriate solution(s).

Module 4: MATH AND DATA ANALYSIS

Lesson 4.7 – Curve Fitting

Learning Objectives

After reading this lesson, you should be able to:

1) *conduct polynomial interpolation using MATLAB,*
2) *conduct spline interpolation using MATLAB,*
3) *regress data to a polynomial using MATLAB.*

What is curve fitting?

Data may be given only at discrete data points. Curve fitting implies techniques to fit a curve to the discrete data and hence be able to find estimates at points other than the given ones. In this lesson, we will limit our discussion to two very common categories of curve fitting: interpolation and regression. One important thing to keep in mind when applying these methods to real-world problems is that they are estimates, and are therefore not guaranteed to be correct. With that said, curve fitting can be a powerful tool for analysis and prediction.

What is interpolation?

Many times, a function, $y = f(x)$ is given only at discrete data points such as, $(x_0, y_0)\,, (x_1, y_1)\,, ..., (x_{n-1}, y_{n-1})\,, (x_n, y_n)$. How does one find the value of y at a value of x that is not one of the given ones? Well, a continuous function $f(x)$ may be used to represent the $(n+1)$ data values with $f(x)$ passing through the $(n+1)$ points. Then one can find the value of y at any other value of x. This is called interpolation. Of course, if x falls outside the range of x values for which the data is given, it is no longer called interpolation but is called extrapolation.

How can I interpolate data in MATLAB?

When programming in MATLAB, the programmer has several functions to help make the difficult task of interpolation an easy one. The two types of interpolation techniques that will be discussed in this lesson are the polynomial and spline interpolation. The MATLAB functions for these models are `polyfit()` and `interp1()`.

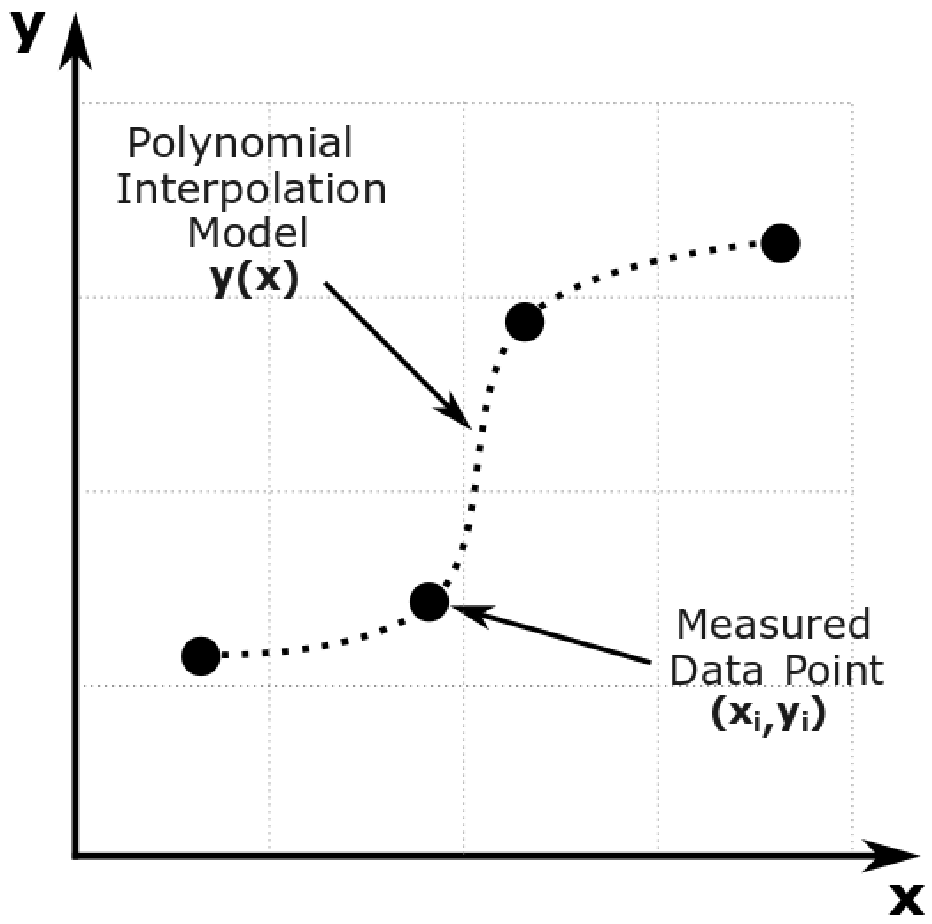


Figure 1: Interpolation of discrete data.

Once the user has input the two vectors of data (x and y, for instance), the `polyfit()` function can be used to interpolate the data to a polynomial function. The `polyfit()` function stores the coefficients of the polynomial in vector form, where they can later be used to generate the polynomial interpolation model. The `polyval()` function uses polynomial coefficients (the output of the `polyfit()` function) to find the interpolated value of y at a chosen value or vector of x.

For interpolation, the order of the polynomial <u>must</u> be exactly one less than the total number of data pairs. So for given data $\left(\mathrm{x_1}, \mathrm{y_1}\right), ..., \left(\mathrm{x_{n+1}}, \mathrm{y_{n+1}}\right)$, the polynomial obtained would be of the form $\mathrm{y} = \mathrm{a_1 x^n} + \mathrm{a_2 x^{n-1}} + ... + \mathrm{a_n}$.

The `polyfit()` function is used to output the coefficients of the polynomial that passes through the data pairs. The output is stored as a vector $[a_1,\ a_2, ..., a_n]$. With these coefficients, the user can symbolically develop the interpolation function and if needed, conduct integration, differentiation, and plotting. Note that the first element corresponds to the coefficient of the highest power (x^n), while the last element corresponds to the constant of the polynomial model.

The `polyval()` function takes the output of the `polyfit()` function and uses it to evaluate the value of the polynomial interpolant at a given value (or a vector) of x. That is, `polyval()` substitutes values for x into the polynomial model. Then `polyval()` returns the corresponding values of y (the predictions) from the polynomial (see Example 1).

Example 1

Using a polynomial model, interpolate the (x, y) data pairs in Table A to a polynomial. Find the value of the interpolant at $x = 4.5$ and output it to the Command Window.

Table A: Data pairs for Example 1.

$\boldsymbol{x}$	1.0	4.0	8.0
$\boldsymbol{y}$	2.2	5.0	7.0

Solution

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To interpolate data to fit a polynomial
fprintf('To interpolate data to fit a polynomial.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Step 1: Inputting raw/known/measured x and y data points
xData = [1   4 8];
yData = [2.2 5 7];
fprintf('The x vector is:\n')
disp(xData)
fprintf('The y vector is:\n')
disp(yData)

%----------------------------- SOLUTION ----------------------------
%Step 2: Choose order of polynomial model
%        Order of the polynomial model, which is # of data points - 1
order = length(xData) - 1;

%Step 3: Finding polynomial model coefficients
%         Outputs coefficients for polynomial model
coef = polyfit(xData,yData,order);

%Step 4: Defining the query value(s)
xQuery   = 6.3;

%Step 5: Predicting a value of y from the polynomial model
yPredict = polyval(coef,xQuery);

%Step 6 (optional): Manually define the interpolation polynomial model
%                   as a symbolic function
syms x
func = coef(1)*x^2 + coef(2)*x + coef(3);
func = vpa(func,3);  %Adjusting precision of the output
```

MATLAB Code (continued) example1.m

```
%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The coefficients of the polynomial model are:\n')
disp(coef)
fprintf('Using these coefficients, we can form the\n')
fprintf('polynomial interpolant y(x) = %s.\n\n',char(func))
fprintf('Using polynomial interpolation (order = %.0f):\n',order)
fprintf('When x = %g, the estimate of y is %g.\n',xQuery,yPredict)
```

We "hardcoded" the polynomial expression in Example 1 for learning efficiency. This way, you can see how a symbolic function can be manually defined from its coefficients (the output of `polyfit()`). See Example 3 for a better method to do this without hardcoding: `poly2sym()`.

Command Window Output Example 1

```
PURPOSE
To interpolate data to fit a polynomial.

INPUTS
The x vector is:
     1     4     8

The y vector is:
    2.2000    5.0000    7.0000

OUTPUTS
The coefficients of the polynomial model are:
   -0.0619    1.2429    1.0190

Using these coefficients, we can form the
polynomial interpolant y(x) = 1.24*x - 0.0619*x^2 + 1.02.

Using polynomial interpolation (order = 2):
When x = 6.3, the estimate of y is 6.39205.
```

What is spline interpolation?

Spline interpolation uses multiple "spline" (math) functions to fit the given data points (Figure 2). Taken as a whole, these splines form a piecewise continuous function: meaning the final model is made up of pieces or splines. Splines can be based on different models, but are commonly linear ($f(x) = a_1x + a_2$) or cubic ($f(x) = a_1x^3 + a_2x^2 + a_3x + a_4$) polynomial functions.

How do I conduct spline interpolation?

When compared to polynomial interpolation, using splines to interpolate the data can prove to be very beneficial in many circumstances. These splines are typically linear or cubic in form and can be implemented in MATLAB using the function `interp1()`.

In some cases, especially with higher order polynomials, a polynomial interpolant can be a bad idea as it may give oscillatory behavior (Figure 4) for otherwise well-behaved smooth functions. When provided a large number of data points, spline interpolation is generally better suited.

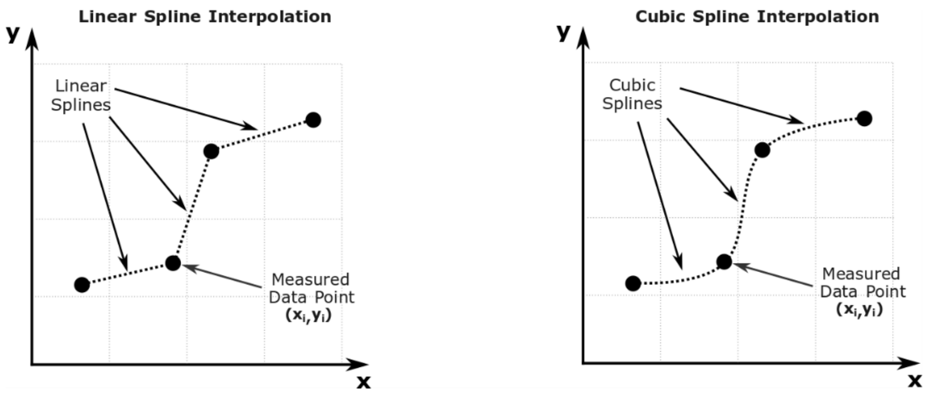


Figure 2: Spline interpolation of discrete data.

Often times when interpolating a data set, a linear spline model is sufficient. In such a case, each data point is connected to the next with a straight line (Figure 2). This technique is commonly used in interpolating data from thermodynamic steam tables. If this is not sufficient, a cubic spline is often used, which connects the data points with cubic functions (nonlinear lines as shown in Figure 2). The MATLAB function, `interp1()`, can be used to interpolate a data set using a specified model (including a linear or cubic-spline model). An example of the usage of this function is: `interp1(xData, yData, xQuery, 'method')`.

The output of the `interp1()` function is a vector of the same size as the input vector of the x value(s). We call these input values "x query" values because they are the values of the independent variable at which we want to make predictions. For example, when $x = 3$, what is the value of y? Here, "$x = 3$" is the query value. Table 1 shows the common interpolation methods that can be used as the input for the `interp1()` function, and Example 2 shows the function in action.

Table 1: Common interpolation models to be used with the `interp1()` function.

Interpolation Method	**Interpolation Model Generated**
`'linear'`	Interpolates via straight lines between each consecutive point (default model).
`'spline'`	Connects each point with a cubic-spline interpolant. The first and second derivatives of the adjoining splines will be continuous.

Example 2

Interpolate the (x,y) data pairs from Table B using linear and cubic spline interpolation. Output the predictions using `fprintf()` at $x = 6.3$.

Table B: Data pairs to be used for Example 2.

x	2.0	5.1	7.7	9.2	10.3
y	1.4	3.3	5.7	10.4	12.5

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To interpolate data by fitting linear and cubic splines
fprintf('To interpolate data by fitting linear and cubic splines.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
%Step 1: Inputting raw/known/measured x and y data points
xData = [2.0 5.1 7.7 9.2 10.3];
yData = [1.4 3.3 5.7 10.4 12.5];
fprintf('The x vector is:\n')
disp(xData)
fprintf('The y vector is:\n')
disp(yData)

%------------------------------ SOLUTION ----------------------------
%Step 2: Defining the query value(s)
xQuery   = 6.3;

%Step 3: Performing spline interpolation
%Predicting y value using linear splines
yLinPredict = interp1(xData,yData,xQuery,'linear');
%Predicting y value using cubic splines
yCubPredict = interp1(xData,yData,xQuery,'spline');

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('Using linear splines:\n')
fprintf('When x = %g, the estimate of y is %g.\n\n',xQuery,yLinPredict)
fprintf('Using cubic splines:\n')
fprintf('When x = %g, the estimate of y is %g.\n\n',xQuery,yCubPredict)
```

The Command Window output shows the predicted y values when $x = 6.5$. These values are fairly different from each other (3.593 for linear splines vs. 4.408 for cubic splines). In the next lesson (Lesson 4.8), you will be able to see more clearly why this is so when we plot the linear and cubic spline functions.

Command Window Output Example 2

```
PURPOSE
To interpolate data by fitting linear and cubic splines.

INPUTS
The x vector is:
    2.0000    5.1000    7.7000    9.2000   10.3000

The y vector is:
    1.4000    3.3000    5.7000   10.4000   12.5000

OUTPUTS
Using linear splines:
When x = 6.3, the estimate of y is 4.40769.

Using cubic splines:
When x = 6.3, the estimate of y is 3.59263.
```

What is regression?

Finding a function that best fits the given data pairs is called regression. When conducting interpolation, all data pairs used must be on the developed curve. On the other hand, a regression curve is not constrained by this requirement. Using MATLAB to develop a regression curve is useful, especially for experimental data, or for developing simplified models.

Let us suppose someone gives you n data pairs:$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, and you want to develop a relationship between the two variables. A simple example is that of measuring stress vs. strain data for a steel specimen under loads lower than the yield point. We expect that the relationship between stress and strain is a straight line. However, because of material imperfections and inaccuracies in data collection, we are not going to get all the data points on a straight line. So, we do the next best thing – draw a straight line that minimizes the sum of the square of the difference between the observed and predicted values (Figure 3). How that is done is a subject for a course in statistics or numerical methods.

In this part of the lesson, we will just concentrate on how to use MATLAB to regress data to polynomials. Although there is a mathematical/statistical difference between polynomial interpolation and regression, there is no explicit difference *in MATLAB syntax* between an interpolation and regression polynomial. Therefore, you should choose the curve fitting method that makes the most sense or gives the best results for your problem.

One of the challenges when fitting some models to a data set is the tendency to *overfit* the data. We will not go into great detail in this lesson, but we want to alert you to this important and common problem. When performing polynomial regression, you should try to choose an order for the polynomial that does not overfit the data.

MATLAB will often display a warning that your polynomial is “badly conditioned” when you are overfitting. Another sign of overfitting is when you have large deviations from your expected curve (see Figure 4). For example, if you had position and time data from an accelerating car, you would not expect to see something like Figure 4 where there is a large deviation from the expected path. Therefore, thinking critically about your results is essential!

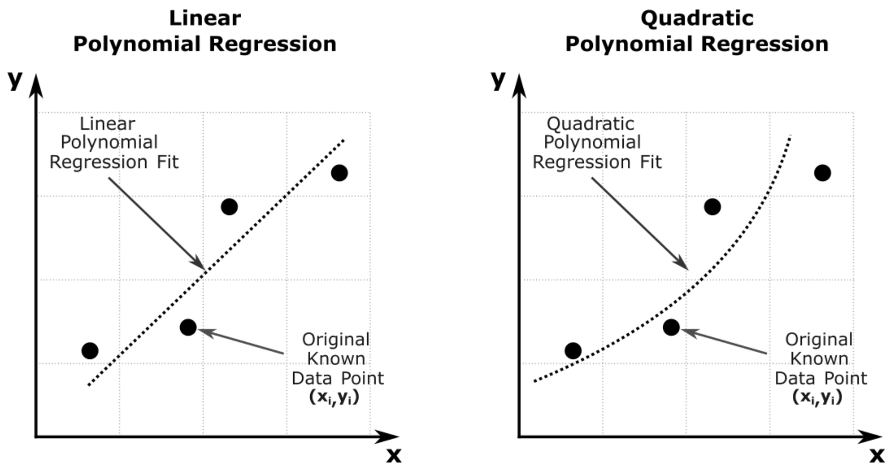

Linear
Polynomial Regression
y
Linear Polynomial Regression Fit
Original Known Data Point (x_i,y_i)
x
Quadratic
Polynomial Regression
y
Quadratic Polynomial Regression Fit
Original Known Data Point (x_i,y_i)
x

Figure 3: Regression of n data points to best fit a given order polynomial.

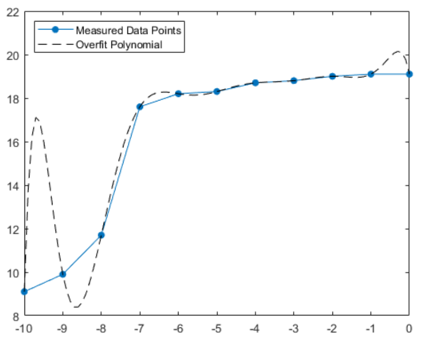


Figure 4: An example of overfitting on position and time data from an accelerating car (code not shown).

How do I do regression in MATLAB?

Similar to interpolation, the first step of making a regression model is to determine the type of function that best fits the data pairs. This lesson will focus on the polynomial regression model, although many other regression models may be used. These other models include exponential, power, and saturation growth models.

To do polynomial regression, you need the following two inputs:

1. Data pairs (x, y)
2. Order of the polynomial of regression, m

For regression, the order of the polynomial chosen must be less than (total number of data pairs minus one). So for given data pairs $(x_1, y_1), ..., (x_n, y_n)$, the polynomial obtained would be of the form $y = a_1 x^m + a_2 x^{m-1} + ... + a_m,\ 1 \leq m \leq n - 2$. Note that for m = n − 1 the regression polynomial would be an interpolating polynomial.

The `polyfit()` function is used to output the coefficients of the regression polynomial. The output is stored as a vector $[a_1, a_2, ..., a_m]$. With these coefficients, the user can symbolically develop the regression function and if needed, conduct integration, differentiation, and plotting. Note that the first element corresponds to the coefficient of the highest power (x^m), while the last element corresponds to the constant of the polynomial model.

The function `polyval()` can be used again for the same purpose as shown in Example 1. In Example 3, it will take the coefficients of a polynomial and x query value(s) as inputs and return the predicted value for y, which it obtains from the regression polynomial.

Example 3

Using MATLAB, regress the given (x, y) data pairs from Table C to a linear and quadratic regression model, and predict the value of y when x is (−300, −100, 20, 125) using both models. Output the predictions and the regression models using `fprintf()` or `disp()`.

x	340	280	200	120	40	40	80
y	2.45	3.33	4.30	5.09	5.72	6.24	6.47

Table C: Data pairs to be used for Example 3.

Solution

MATLAB Code example3.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To regress data to best fit a polynomial model
fprintf('To regress data to best fit a polynomial model.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Step 1: Inputting raw/known/measured x and y data points
xData = [-340 -280 -200 -120 -40 40 80];
yData = [2.45 3.33 4.30 5.09 5.72 6.24 6.47];
fprintf('The x vector is:\n')
disp(xData)
fprintf('The y vector is:\n')
disp(yData)
```

MATLAB Code (continued) example3.m

```matlab
%----------------------------- SOLUTION ----------------------------
%Step 2: Choose order of polynomial model(s)
linOrder  = 1;  %Defining "1" as the order of the linear polynomial
quadOrder = 2;  %Defining "2" as the order of the quadratic polynomial

%Step 3: Finding polynomial model coefficients
linCoef  = polyfit(xData,yData,linOrder);
quadCoef = polyfit(xData,yData,quadOrder);

%Step 4: Defining the query value(s)
xQuery   = [-300 -100 20 60];  %Want to predict y at all these x values

%Step 5: Predicting a value of y from the polynomial model
yLinPredict = polyval(linCoef,xQuery);
yQuadPredict = polyval(quadCoef,xQuery);

%Step 6: Define the regression polynomial as a symbolic function
syms x     %Defining the symbolic variable "x"
%Defining the regression models as a symbolic functions
linFunc  = poly2sym(linCoef,x);
quadFunc = poly2sym(quadCoef,x);

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The linear regression polynomial is:\n')
fprintf('    y(x) = %s.\n',char(vpa(linFunc,3)))
fprintf('The quadratic regression polynomial is:\n')
fprintf('    y(x) = %s.\n\n',char(vpa(quadFunc,3)))

fprintf('Using linear polynomial regression, the y estimates\n')
fprintf(' (corresponding to xQuery) are:\n')
disp(yLinPredict)
fprintf('Using quadratic polynomial regression, the y estimates\n')
fprintf(' (corresponding to xQuery) are:\n')
disp(yQuadPredict)
```

Command Window Output Example 3

```
PURPOSE
To regress data to best fit a polynomial model.

INPUTS
The x vector is:
  -340  -280  -200  -120   -40    40    80

The y vector is:

    2.4500    3.3300    4.3000    5.0900    5.7200    6.2400    6.4700
```

Command Window Output (continued) Example 3

```
OUTPUTS
The linear regression polynomial is:
    y(x) = 0.00939*x + 5.95.
The quadratic regression polynomial is:
    y(x) = 0.00628*x - 1.22e-5*x^2 + 6.02.
Estimate the value of y(x) at x values of:
  -300  -100    20    60

Using linear polynomial regression, the y estimates
 (corresponding to xQuery) are:
    3.1370    5.0146    6.1411    6.5167

Using quadratic polynomial regression, the y estimates
 (corresponding to xQuery) are:
    3.0386    5.2716    6.1423    6.3544
```

In Example 3, since we are inputting a vector of values to `polyval()` (using the variable `xQuery`), it will return a vector of predictions to us, which can be seen in the Command Window output. Remembering the inputs and outputs of these curve fitting functions is essential to proper implementation.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Polynomial interpolation	`polyfit()`	`polyfit(x,y,order)`

Task	Syntax	Example Usage
Polynomial regression	`polyfit()`	`polyfit(x,y,order)`
Spline interpolation	`interp1()`	`interp1(x,y,xQuery,'method')`
Convert polynomial coefficients to symbolic function form	`poly2sym()`	`poly2sym(coef,x)`

Multiple Choice Quiz

(1). The MATLAB function used to find the coefficients of a polynomial interpolation or regression model for given data pairs is

(a) `polyfit()`

(b) `polyval()`

(c) `interp1()`

(d) `interceof()`

(2). The result of the curve fitting procedure completed in the following program is

```
clc
clear
time  = [0 2 3];
vel   = [0 4 6];
time1 = 2.5;
vel1  = interp1(time,vel,time1,'linear');
vel1
```

(a) polynomial interpolation

(b) spline interpolation

(c) polynomial regression

(d) None of the above

(3). The output of the last line is

```
clc
clear
time  = [0 2 3];
vel   = [0 4 6];
time1 = 2.5;
vel1  = interp1(time,vel,time1,'linear');
vel1
```

(a) 2.5

(b) 5.0

(c) 7.0

(d) 10.0

(4). Complete the code to output the regression model as a symbolic function.

```
clc
clear
xd = [0 3 5];
yd = [0 4 8];
syms x
```

(a) `coef = polyfit(xd,yd,1);y = coef(2)*x + coef(1)`

(b) `coef = polyfit(yd,xd,1);y = coef(2)*x + coef(1)`

(c) `coef = polyfit(xd,yd,1);y = coef(1)*x + coef(2)`

(d) `coef = polyfit(yd,xd,1);y = coef(1)*x + coef(2)`

(5). The function that uses previously found coefficients of a polynomial interpolant as an input to calculate the value of the function at a given point is

(a) `polyfit()`

(b) `polyval()`

(c) `interp1()`

(d) `intereval()`

Problem Set

(1). Given are (x, y) data pairs in Table A.

Table A: Data pairs for Exercise 1.

$\boldsymbol{x}$	1.4	2.3	5.0	7.5
$\boldsymbol{y}$	3.2	1.7	6.1	3.8

Complete the following.

(a) Interpolate the data using a polynomial interpolant. Find the value of y when $x = 4.75$.

(b) Interpolate the data using linear spline interpolation. Find the value of y when $x = 4.75$.

(c) Interpolate the data using cubic-spline interpolation. Find the value of y when $x = 4.75$.

(2). The upward velocity of a rocket is given as a function of time in Table B.

Table B: Upward rocket velocity at a given time.

t (s)	0	10	15	20	22.5	30
v(t) m/s	0	227.04	362.78	517.35	602.97	901.67

Using MATLAB, complete the following.

(a) Using a polynomial interpolant, find velocity as a function of time.

(b) Find the velocity at $t = 16$ s.

(3). A curve needs to be fit through the seven points given in Table C to fabricate the cam. The geometry of a cam is given in Figure A.

Each point on the cam shown in Figure A is measured from the center of the input shaft. Table C shows the x and y measurement (inches) of each point on the camshaft.

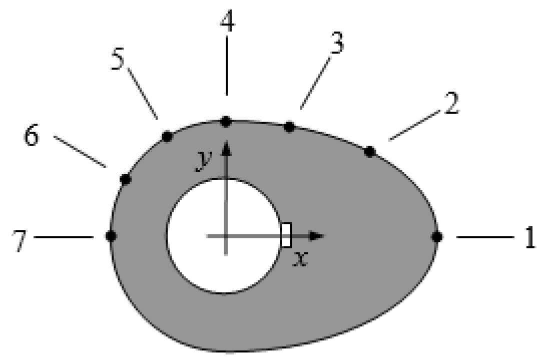


Figure A: Schematic of cam profile

Table C: Geometry of the cam corresponding to Figure A.

Point	x (in)	y (in)
1	2.20	0.00
2	1.28	0.88
3	0.66	1.14
4	0.00	1.20
5	-0.60	1.04
6	-1.04	0.60
7	-1.20	0.00

Using MATLAB, find a smooth curve that passes through all seven data points of the cam. Output this model to the Command Window.

(4). Using MATLAB, regress the following (x, y) data pairs (Table D) to a linear polynomial and predict the value of y when $x = 55, 20, -10$.

Table D: Data pairs (x, y) for Exercise 1.

x	y
325	2.6
265	3.8
185	4.8
105	5.0
25	5.72
– 55	6.4
– 70	7.0

Use the `fprintf()` and/or the `disp()` functions to output the regression model and the predictions to the Command Window.

(5). To simplify a model for a diode, it is approximated by a forward bias model consisting of DC voltage, V_d, and resistor, R_d. Below is the collected data of current vs. voltage for a small signal (Table E).

Table E: Current versus voltage for a small signal.

V (volts)	I (amps)
0.6	0.01
0.7	0.05
0.8	0.20

V (volts)	I (amps)
0.9	0.70
1.0	2.00
1.1	4.00

Regress the data in Table E to a linear model of the voltage as a function of current. Approximate the voltage when 0.35 amps of current is applied to the diode and output this result using `fprintf()`.

(6). To find contraction of a steel cylinder, one needs to regress the thermal expansion coefficient data to temperature. The data is given below in Table F.

Table F: The thermal expansion coefficient at given temperatures

Temperature, T ($^{\circ}F$)	Coefficient of thermal expansion, α (in/in/$^{\circ}F$)
80	6.47×10^{-6}
40	6.24×10^{-6}
– 40	5.72×10^{-6}
– 120	5.09×10^{-6}
– 200	4.30×10^{-6}
– 280	3.33×10^{-6}
– 340	2.45×10^{-6}

Find the coefficient of thermal expansion when the temperature is $-150^{\circ}F$ using

(a) linear polynomial regression,

(b) quadratic polynomial regression, and

(c) cubic spline interpolation.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.8 – Curve Fitting: Plotting the Results

Learning Objectives

After reading this lesson, you should be able to:

1) *plot polynomial interpolation and regression models,*
2) *plot spline interpolation models.*

How can I plot the results of curve fitting?

In Lesson 4.7, we covered how to perform some curve fitting methods on discrete data using MATLAB. In this lesson, we further explore the application of curve fitting methods by looking at how they can be visualized using MATLAB.

To plot any function, we need to find its value across the domain we want to plot. To do this for a curve fitting example, the programmer needs a model that fits the given data pairs, a vector (domain) to plot the model on (query values), and a vector of predicted values corresponding to the desired domain of the plot.

From the last lesson, we know that we can use `polyval()` for both polynomial interpolation and regression models to find their values at given values of x. Example 1 shows how to plot polynomial models that are found using `polyfit()`. This process does not require any new functions or syntax; although, there are a few things to watch out for when completing this process on your own. Be careful to clearly identify the given values from the predicted ones. Likewise, always make sure that you are correctly choosing and referencing the x query values for plotting.

Example 1

Fit a fifth-order polynomial regression model to the data pairs given in Table A, and plot the resulting model and the given data pairs on one plot. Include a title, axis labels, and legend.

Table A: The data pairs used in Example 1.

$\boldsymbol{x}$	0	1	2	3	4	5	6	7	8
$\boldsymbol{y}$	1.00	0.540	-0.416	-0.990	-0.654	0.284	0.960	0.754	-0.146

Solution

MATLAB Code example1.m

```
clc
clear
close all

%------------------------------ PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To plot the results of polynomial models
fprintf('To plot the results of polynomial models.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
%Step 1: Inputting raw/known/measured x and y data points
xData = [0 1 2 3 4 5 6 7 8];
yData = [1.00, 0.540, -0.416, -0.990, -0.654, 0.284, 0.960,...
         0.754, -0.146];

%------------------------------ SOLUTION ----------------------------
%Step 2: Choose order of polynomial model
order = 6;
fprintf('Regress the data to a polynomial model (order = %.0f).\n\n',order)

%Step 3: Finding polynomial model coefficients
coef = polyfit(xData,yData,order);  %Outputs coefs for polynomial

%Step 4: Defining the query value
xQuery    = pi;  %Defining a new x vector to plot

%Step 5: Predicting a value of y from the polynomial model
yPredict = polyval(coef,xQuery);

%Step 6: Finding points (of models) to plot
xQueryPlot   = xData(1):0.1:xData(end);   %Defining a new x vector to plot
yPredictPlot = polyval(coef,xQueryPlot);  %Plugging new x query values into
                                          %    polynomial from polyfit()
%Step 7a: Plotting
figure                                     %Creating a blank figure
plot(xData,yData,'o')                      %Plotting original data points
hold on
grid on
plot(xQueryPlot,yPredictPlot,'r','LineWidth',2)  %Plotting the model
hold off

%Step 7b: Figure formatting
title('Regression Model')
xlabel('x')
ylabel('y')
legend('original data points','polynomial model')

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('When x = %g, the estimate for y is %g.\n',xQuery,yPredict)
```

```
Command Window Output                                              Example 1

PURPOSE
To plot the results of polynomial models.

INPUTS
Regress the data to a polynomial model (order = 6).

OUTPUTS
When x = 3.14159, the estimate for y is -0.983931.
```

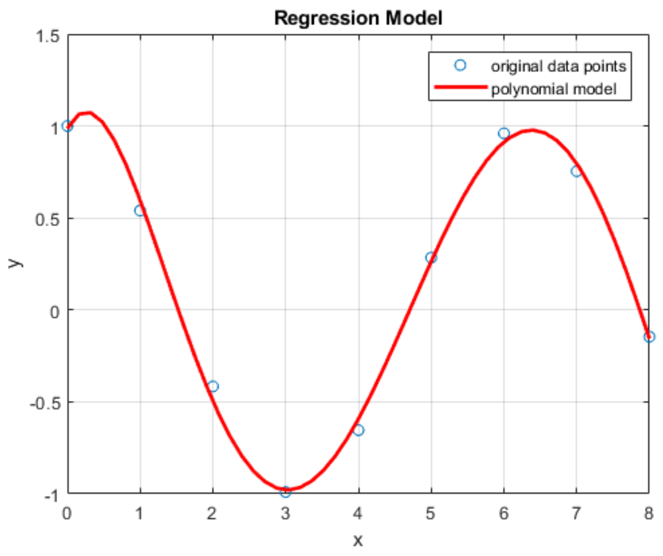


Figure 1: Figure output for Example 1 showing the plot of a regression model.

The raw y data points in Example 1 were generated from $f(x) = \cos(x)$, so the exact answer for `yEstimate` would be `cos(pi)=-1`. We can see that our estimate is relatively close when comparing the discrete values at `cos(pi)` (-1 vs. -0.973) and the original data points compared to the regression model we found (Figure 1). Note that we would not know the function $f(x) = \cos(x)$ in a real world scenario. If we did, there would be no reason to use curve fitting methods to find it!

In Example 2, we show an example of plotting spline interpolation models. This requires less steps than plotting the polynomial models, but the basics are the same. `interp1()` returns a vector of values corresponding to the query points, so we can plot the results directly without any further substitution needed.

Example 2

Fit linear and cubic-spline interpolation models to the data pairs given in Table B, and plot the resulting models and the given data pairs on one plot for comparison. Include a title, axis labels, and legend.

Table B: The data pairs used in Example 2.

x	0	1	2	3	4	5	6	7	8
y	1.00	0.540	-0.416	-0.990	-0.654	0.284	0.960	0.754	-0.146

Solution

MATLAB Code example1.m

```
clc
clear
close all

%------------------------------ PURPOSE ------------------------------
%To plot the results of spline interpolation models

%------------------------------ INPUTS -------------------------------
%Step 1: Inputting raw/known/measured x and y data points
xData = [0 1 2 3 4 5 6 7 8];
yData = [1.00, 0.540, -0.416, -0.990, -0.654, 0.284, 0.960,...
         0.754, -0.146];

%------------------------- SOLUTION/OUTPUTS --------------------------
%Step 2: Defining the query values
%This is a good method for defining x query values for plotting because it
%    will automatically change if xData changes.
domainSpread = max(xData) - min(xData);
xQuery = min(xData):domainSpread/100:max(xData);

%Step 3: Using splines to interpolate data
yEstimate1 = interp1(xData,yData,xQuery,'linear');  %Linear splines
yEstimate2 = interp1(xData,yData,xQuery,'spline');  %Cubic splines from
polyfit

%Step 4a: Plotting
figure                                         %Create a figure
plot(xData,yData,'oc','MarkerFaceColor','c')   %Plotting original data
points
hold on
plot(xQuery,yEstimate1,'-.r','LineWidth',2);   %Plotting linear splines
plot(xQuery,yEstimate2,'k','LineWidth',2);     %Plotting cubic splines
hold off

%Step 4b: Figure formatting
title('Comparing Splines');
xlabel('x')
ylabel('y')
legend('data points','linear','cubic')
```

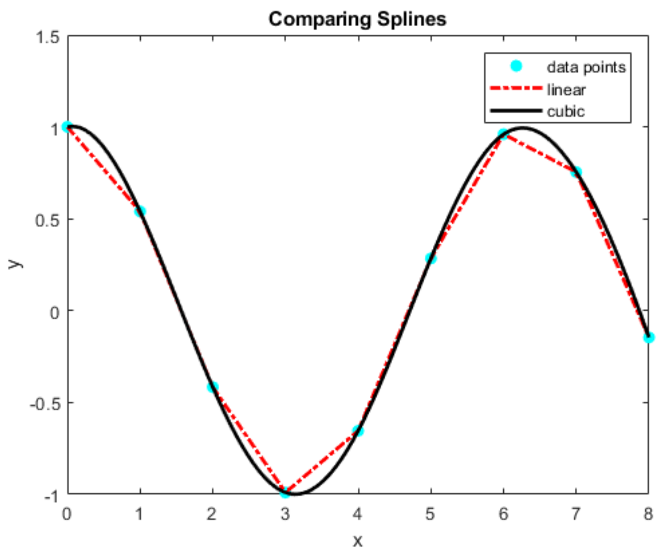


Figure 2: Figure output for Example 2 comparing linear and cubic spline interpolation results.

What are some common mistakes when plotting curve fitting results?

Some common mistakes made when first learning to perform curve fitting and plotting the results are:

(1) Plotting the coefficients of a polynomial (the direct output from `polyfit()`)

(2) Using the given x values as the x query values

(3) Plotting the given x values vs. the predicted y values

(4) Using the wrong function: `polyfit()`, `polyval()`, `interp1()`

To elaborate on the mistakes listed above, plotting the coefficients of a polynomial is incorrect as this is not a prediction: it is just part of a polynomial model. One must substitute x query values into that polynomial to get the predictions to plot.

Using the given x values as the x query values is incorrect because the whole purpose of curve fitting is to find out more information than we already know. Therefore, we should choose x query values for which we do not already know the corresponding y values.

Plotting the given x values vs. the predicted y values is wrong because it is mathematically false. If we predict the value of $y(2.5)$ with our model, we cannot then automatically say that $y(2.5) = y(1)$, which is exactly what is happening when you plot predicted y values against an incorrect x domain. Another symptom of this same problem is when MATLAB returns an error for the plotted x and y vectors being unequal lengths. This error cannot occur if you properly calculated and selected your x query and y predicted values. So, that is where you should start looking if this error occurs when plotting predicted values.

Finally, keep in mind that we have covered only three functions to do all of the different interpolation and regression methods. Those functions are `polyfit()`, `polyval()`, and `interp1()`. `polyfit()` and `polyval()` pertain to polynomial interpolation and regression. For spline interpolation (both linear and cubic-splines), use `interp1()`.

Multiple Choice Quiz

(1). To plot a polynomial regression or interpolation model, one does not need to use the function

(a) `polyfit()`

(b) `polyval()`

(c) `plot()`

(d) `interp1()`

(2). The default output of `interp1()` is/are

(a) polynomial model coefficients

(b) predicted values corresponding to the provided x query values

(c) a polynomial model in symbolic form

(d) none of the above

(3). Complete the code to prepare the plot of the interpolation model.

```
clc
clear

x  = [1 3 5 7 10];
y  = [2 5 6 8 13];
n  = length(x)-1;
xx = min(x):0.1:max(x);
```

(a) `coef = polyfit(x,y,n);yy = polyval(xx,coef);`

(b) `coef = polyfit(x,y,n);yy = polyval(coef,xx);`

(c) `coef = polyfit(y,x,n);yy = polyval(xx,coef);`

(d) `coef = polyfit(y,x,n);yy = polyval(coef,xx);`

(4). The output of `polyfit()` is/are

(a) polynomial model coefficients

(b) predicted values corresponding to the provided x query values

(c) a polynomial model in symbolic form

(d) none of the above

(5). The most appropriate plotting call for visualizing a curve fitting result (where `xGiven` = given x values, `xPredict` = predicted x values, `yGiven` = given y values, `yPredict` = predicted y values) is

(a) `plot(xGiven,yPredict)`

(b) `plot(xQuery,yGiven)`

(c) `plot(xQuery,yPredict)`

(d) `plot(xGiven,yGiven)`

Problem Set

(1). A curve needs to be fit through the seven points given in Table A to fabricate the cam. The geometry of a cam is given in Figure A.

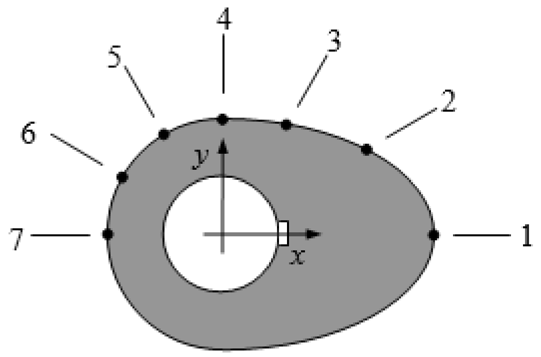


Figure A: Schematic of the cam profile

Each point on the cam shown in Figure A is measured from the center of the input shaft. Table A shows the x and y measurement (inches) of each point on the camshaft.

Table A: Geometry of the cam corresponding to Figure A.

Point	x (in)	y (in)
1	2.20	0.00
2	1.28	0.88
3	0.66	1.14
4	0.00	1.20
5	-0.60	1.04
6	-1.04	0.60
7	-1.20	0.00

Using MATLAB, output a smooth curve that passes through all seven data points of the cam. Plot this model (with a black line) and the given data points (with cyan circles) on the same plot. Include a title, axis labels, and legend.

(2). A vegetable processing company has finished their first prototype of a new carrot cutting tool, and needs your help with the analysis. The device (shown in Figure B) consists of razor blade attached to a spring loaded hinged arm. The arm is free to rotate about the hinge from 0° to 180°, as shown. The spring is fully loaded when the arm is at the 0° mark and has a neutral position at the 180° mark of the arm.

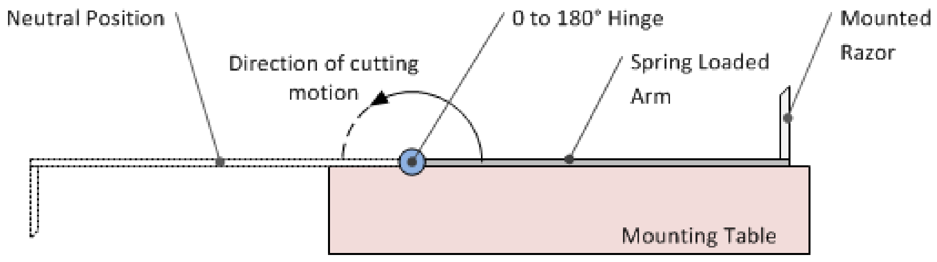


Figure 3: diagram112e_springloadedcutting

Figure B: Cutting device for Exercise 4.

In a laboratory, the force exerted on the far end of the arm is measured at several angles as the arm rotates about the hinge. The results are recorded in Table B.

Table B: Measured force at a given angle.

Angle (°)	0.0	20.5	70.1	160.5	180.0
Force (lbs)	12.20	10.56	7.65	2.01	0.00

If the arm (C) measures 8.75 inches from the hinge to the razor, plot the torque (ft-lbs) exerted by the spring as the arm swings from the fully-loaded to fully-neutral position (use a cubic-spline interpolation model). Also, predict the value of the torque once the arm has swung 90°.

Hint: Torque = Force × Arm length. Be aware of the units.

(3). To maximize a catch of bass in a lake, it is suggested to throw the line to the depth of the thermocline. The characteristic feature of the thermocline is the sudden change in temperature. We are given the temperature vs. depth data for a lake in the table below (Table C).

Table C: Water Temperature at a given depth.

Temperature, T ($°C$)	Depth, z (m)
19.1	0
19.1	-1
19.0	-2
18.8	-3
18.7	-4
18.3	-5
18.2	-6
17.6	-7
11.7	-8
9.9	-9
9.1	-10

Conduct polynomial and cubic spline interpolation on the data, and plot the two interpolants. From the plot, can you judge where the sudden change in temperature is taking place?

(4). Using MATLAB, regress the following (x, y) data pairs (Table D) to a linear polynomial and predict the value of y when $x = 55, 20, -10$.

Table D: Data pairs (x, y) for Exercise 1.

x	y
325	2.60

x	y
265	3.80
185	4.80
105	5.00
25	5.72
-55	6.40
-70	7.00

Use the `fprintf()` and/or the `disp()` functions to output the regression model results to the Command Window. Plot the regression model with data points (given and predicted) on a (x, y) linear plot.

(5). To find contraction of a steel cylinder, one needs to regress the thermal expansion coefficient data to temperature. The data is given below in Table E.

Table E: The thermal expansion coefficient at given temperatures

Temperature, T ($^{\circ}F$)	Coefficient of thermal expansion, α(in/in/$^{\circ}F$)
80	6.47×10^{-6}
40	6.24×10^{-6}
-40	5.72×10^{-6}
-120	5.09×10^{-6}
-200	4.30×10^{-6}
-280	3.33×10^{-6}
-340	2.45×10^{-6}

Fit the above data to $\alpha = a_0 + a_1 T + a_2 T^2$. Plot the regression model along with the given data points on one plot. Include a grid, axis labels, and a legend.

Module 4: MATH AND DATA ANALYSIS

Lesson 4.9 – Ordinary Differential Equations

Learning Objectives

After reading this lesson, you should be able to:

1) *solve an ordinary differential equation using MATLAB.*

What is a differential equation?

An equation that consists of derivatives is called a differential equation. Differential equations have applications in all areas of science and engineering. A mathematical formulation of most physical and engineering problems leads to differential equations. So, it is important for engineers and scientists to know how to set up differential equations and solve them. Differential equations are of two types:

1. ordinary differential equations (ODE) and
2. partial differential equations (PDE).

An ordinary differential equation is that in which all the derivatives are with respect to a single independent variable. Examples of ordinary differential equations include:

a. $\dfrac{d^2y}{dx^2} + 2\dfrac{dy}{dx} + y = 0,\ \dfrac{dy}{dx}(0) = 2,\quad y(0) = 4,$

b. $\dfrac{d^3y}{dx^3} + 3\dfrac{d^2y}{dx^2} + 5\dfrac{dy}{dx} + y = \sin x,\ \dfrac{d^2y}{dx^2}(0) = 12\ \dfrac{dy}{dx}(0) = 2,\ y(0) = 4$

c. $\dfrac{d^2y}{dx^2} - 5y = 6(100 - x),\ y(0) = 5,\ y(100) = 27$

Ordinary differential equations are classified in terms of order and degree. The *order* of an ordinary differential equation is the same as the highest derivative and the *degree* of an ordinary differential equation is the power of the highest derivative. Thus, the differential equation,

$$x^3\frac{d^3y}{dx^3} + x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + xy = e^x$$

is of order 3 and degree 1, whereas the differential equation

$$\left(\frac{dy}{dx} + 1\right)^2 + x^2\frac{dy}{dx} = \sin x$$

is of order 1 and degree 2.

How do I set up and solve a differential equation?

An engineer's approach to differential equations is different from that of a mathematician. While the latter is interested in the mathematical solution, an engineer should be able to interpret and implement the result physically. So, an engineer's approach can be divided into three phases:

1. formulation of a differential equation from a given physical situation,
2. solving the differential equation using given conditions, and
3. interpreting the results physically for implementation.

To solve a differential equation, we need

1. the differential equation, `deq`,
2. the initial/boundary condition(s), `IV`, and
3. the independent variable, `x`.

In MATLAB, the function to solve an ordinary differential equation exactly is `dsolve()`. A usage example is `dsolve(deq,IV,x)`. The number of initial or boundary conditions needed is the same as the order of the differential equation.

Example 1

Analytically solve the following differential equation using MATLAB.

$$2\frac{dy}{dx} = -3y + 5e^x, \quad y(0) = 5$$

Solution

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To solve a first order ODE
fprintf('To solve a first order ODE.\n\n')
```

MATLAB Code (continued) example1.m

```
%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
syms y(x)
deq = 2*diff(y,x) == -3*y + 5*exp(x);    %Defining the ODE
fprintf('The equation to solve is: %s\n',char(deq))

IV  = y(0)==5;  %Defining the initial value of the dependent variable, y
fprintf('With the intial value: %s\n\n',char(IV))

%------------------------------ SOLUTION -----------------------------
deSoln = dsolve(deq,IV,'x');  %Solving the ODE with initial values
deSoln = simplify(deSoln);    %Algebraically simplifying the solution

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The solution of the ODE is %s\n',char(deSoln))
```

```
Command Window Output                                        Example 1
PURPOSE
To solve a first order ODE.

INPUTS
The equation to solve is: 2*diff(y(x), x) == 5*exp(x) - 3*y(x)
With the intial value: y(0) == 5

OUTPUTS
The solution of the ODE is 4*exp(-(3*x)/2) + exp(x)
```

Note in Example 1 that the $\frac{dy}{dx}$ part of the equation is entered as `diff(y,x)`. This is possible because we have defined the symbolic function `y(x)`. Therefore, both `y` and `x` are defined as symbolic variables in MATLAB. Also, note the use of the double equals (`==`) to separate the left- and right-hand sides of the differential equation. Recall that we used this same syntax when entering a left- and right-hand side of an equation when solving for its roots in Lesson 4.3. This must be done as MATLAB always associates a single equals sign (`=`) with assigning a value to a variable, which we have already done at the beginning of that line (`deq =`). In other words, when entering an equation with a left- and right-hand side, use double equals.

The `simplify()` function, in Example 1, is used to make the output aesthetically pleasing. It will try to algebraically simplify a mathematical expression that you give it as an input. Look at the code without `simplify()` to see the difference!

How do I solve a higher order ODE?

The `dsolve()` function is used to solve ordinary differential equations of all orders. However, you must make sure to place all the initial and boundary conditions correctly into the function. Look at Example 2 that shows how to solve a typical second order ordinary differential equation.

Example 2

Using the `dsolve()` function, solve the following ordinary differential equation.

$$3\frac{d^y}{dx^2} + 5\frac{dy}{dx} + 7y = 11e^{-13x}, \quad y(0) = 19, \quad \frac{dy}{dx} = 17$$

Solution

When inputting the equation, look at how the $\frac{d^2y}{dx^2}$ part of the equation is entered as `diff(y,x,2)`. Other than that, the only changes made to the `dsolve()` function when compared to Example 1 are the number of initial values used to solve the differential equation. Since this ODE is second order, we require two initial values.

The solution to this differential equation is rather long, and one can see that using MATLAB to solve it saved some time and effort.

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To solve a 2nd order ODE
fprintf('To solve a 2nd order ODE.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
syms y(x) Dy
deq = 3*diff(y,x,2) + 5*diff(y,x) + 7*y == 11*exp(-13*x);  %Defining ODE
fprintf('The equation to solve is:\n')
fprintf('%s\n\n',char(deq))

%Defining the initial values for the 2nd order ODE
Dy = diff(y,x);                %Defining Dy for IV
IV = [y(0)==9, Dy(0)==17];
fprintf('With the intial conditions:\n')
fprintf('%s\n\n',char(IV))

%------------------------------ SOLUTION -----------------------------
deSoln = dsolve(deq,IV,'x');        %Solving the ODE with initial values
deSoln = vpa(simplify(deSoln),4);   %Simplifying the solution
```

MATLAB Code (continued) example2.m

```
%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The solution of the ODE is:\n')
fprintf('%s\n',char(deSoln))
```

```
Command Window Output                                            Example 2
PURPOSE
To solve a 2nd order ODE.

INPUTS
The equation to solve is:
7*y(x) + 5*diff(y(x), x) + 3*diff(y(x), x, x) == 11*exp(-13*x)

With the intial conditions:
matrix([[y(0) == 9, subs(diff(y(x), x), x, 0) == 17]])

OUTPUTS
The solution of the ODE is:
0.0245*exp(-13.0*x) + 8.976*exp(-0.8333*x)*cos(1.28*x)
      + 19.37*exp(-0.8333*x)*sin(1.28*x)
```

What are the limitations of using the `dsolve()` function?

As a programmer, one of the main pitfalls that you may experience is that the `dsolve()` function may not output a solution to a given differential equation. The `dsolve()` function uses symbolic manipulations to solve differential equations, and although very advanced, the algorithm cannot solve all ordinary differential equations (most do not have explicit solutions). You may receive this warning message in the Command Window:

```
Warning: Unable to find explicit solution.
```

If this is the case, solve the differential equation using a numerical method. Two (of the many) MATLAB functions that numerically solve a differential equation are `ode45()` and `ode23()`. For more information on using these functions, use the MATLAB documentation.

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Solve a differential equation analytically	`dsolve()`	`dsolve(deq,IV,x)`

Numerically solve a differential equation	`ode45()`	`ode45(deq,tspan,IV)`
Numerically solve a differential equation	`ode23()`	`ode23(deq,tspan,IV)`

Multiple Choice Quiz

(1). The MATLAB function for solving an initial value ordinary differential equation is

(a) `ode()`

(b) `diff()`

(c) `dsolve()`

(d) `diffsolve()`

(2). The most appropriate choice for defining the following differential equation

$$7\frac{dy}{dt} + 3y = 4, \quad y(0) = 2$$

for use with the `dsolve()` function is

(a) `deq = 7diff(y,x) + 3*y == 4`

(b) `deq = 7Dy + 3y == 4`

(c) `deq = 7* diff(y,x) + 3*y == 4`

(d) `deq = 7diff(y,x) + 3*y = 4`

(3). Complete the code to solve the initial value ordinary differential equation.

```
clc
clear

syms y(x)
eqn1 = diff(y,x) == cos(x);
inCond1 = y(0) == 0;
```

(a) `dsolve(eqn1,inCond1,'x')`

(b) `dsolve(eqn1,inCond1,x)`

(c) `dsolve(eqn1,inCond1,'y')`

(d) `dsolve(inCond1,eqn1,'x')`

(4). To solve

$$\frac{d^2y}{dx^2} = 5x(30-x), \quad y(0) = 5, \quad \frac{dy}{dx}(30) = 7$$

the most appropriate MATLAB line of code to add to the following program is

```
clc
clear

syms y(x)
Dy = diff(y,x)
IV = [y(0)==5, Dy(30)==7]
```

(a) `dsolve(diff(y,x,2) == 5*x*(30-x), IV, x)`

(b) `dsolve(diff(y,x,2) == 5*x*(30-x), IV, 'y')`

(c) `dsolve(D2y == 5*x*(30-x), IV, 'x')`

(d) `dsolve(D2y == 5*x*(30-x), IV)`

(5). A MATLAB function that finds the numerical solution to an ordinary differential equation is

(a) `ode45()`

(b) `oda555()`

(c) `dsolve()`

(d) `trapz()`

Problem Set

(1). Solve the following initial value differential equation in MATLAB.

$$3\frac{dy}{dx} + 6y = e^{-x}, \quad y(0) = 6$$

.

Find $y(10)$.

(2). Solve the following initial value differential equation in MATLAB.

$$7\frac{d^2y}{dx^2} + 11\frac{dy}{dx} + 13y = \sin(x), \quad y(0) = 6, \quad y'(0) = 17$$

Find $y(10)$ and $y'(10)$. Plot y as a function $x$from $x = 0$ to $x = 20$.

(3). Solve the following boundary value differential equation in MATLAB.

$$\frac{d^2y}{dx^2} - 3 \times 10^{-6}y = 7.5 \times 10^{-7}(100 - x), \quad y(0) = 0, \quad y(100) = 0$$

Find $y(15)$, and the maximum value of y.

(4). A ball bearing at $1200K$ is allowed to cool down in air at an ambient temperature of $300K$. Assuming heat is lost only due to radiation, the differential equation for the temperature of the ball is given by

$$\frac{d\theta}{dt} = -2.2067 \times 10^{-12}\left(\theta^4 - 81 \times 10^8\right), \quad \theta(0) = 1200K$$

where θ is in K and t in seconds. Using MATLAB, find the temperature, the rate of change of temperature, and the rate at which heat is lost at $t = 480$ seconds. The rate at which the heat is lost (in Watts) is given by

$$\text{Rate at which heat is lost} = 2.42 \times 10^{-10}\left(\theta^4 - 81 \times 10^8\right)$$

(5). Pollution in lakes can be a serious issue as they are used for recreational use. One is generally interested in knowing that if the concentration of a particular pollutant is above acceptable levels. The differential equation governing the concentration of pollution in a lake as a function of time is given by

$$25 \times 10^6\frac{dC}{dt} + 1.5 \times 10^6 C = 0$$

If the initial concentration of the pollutant is 10^7parts$/m^3$, and the acceptable level is 5×10^6parts$/m^3$, how long will it take for the pollution level to decrease to an acceptable level? Plot the concentration of the pollutant as a function of time from the initial time to the time it takes the pollution level to decrease to the acceptable level.

(6). The speed (rad/s) of a motor without damping for a voltage input of 20 V is given by

$$20 = (0.02)\,\frac{dw}{dt} + (0.06)w$$

If the initial speed is zero $(w(0) = 0)$, what is the speed of the motor at $t = 0.8s$? What is the angular acceleration of the motor at $t = 0.8s$.

(7). For a solid steel shaft to be shrunk-fit into a hollow hub, the solid shaft needs to be contracted. Initially at a room temperature of 27 ^oC, the solid shaft is placed in a refrigerated chamber that is maintained at -33 ^oC. The differential equation governing the change of temperature of the solid shaft θ is given by

$$\frac{d\theta}{dt} = -5.33 \times 10^{-6}\begin{pmatrix} -3.69 \times 10^{-6}\theta^4 + 2.33 \times 10^{-5}\theta^3 + 1.35 \times 10^{-3}\theta^2 \\ +5.42 \times 10^{-2}\theta + 5.588 \end{pmatrix}(\theta + 33)$$

Using MATLAB, find the temperature and the rate of change of temperature after the steel shaft has been in the chamber for 12 hours.

Module 5: CONDITIONAL STATEMENTS

Lesson 5.1 – Conditions and Boolean Logic

Learning Objectives

After reading this lesson, you should be able to:

- *identify different relational operators,*
- *construct logical expressions,*
- *perform data type identification with MATLAB functions,*
- *round up, round down, and round numbers to integers.*

What are conditions?

Conditions are simply logical expressions: they are not unique to programming. You are likely familiar with relational operators like $<$, $\leq$, $>$, etc. (although the syntax may be slightly different in MATLAB). This is the basic syntax we use to create conditions in MATLAB. These conditions are either true or false. Either `4 < 5` (four is less than five) or it is not. Note the last two operators seen in Table 1 can also be used with non-numeric values like text. That is, you cannot ask if one word is quantitatively greater than another, but you can ask if they are the same word or not. Note the conditional operator for *comparing* two values to see if they are equal (==) is *not* the same as *setting* a variable equal to a value (=).

Table 1: Relational operators in MATLAB and what they mean.

Logical Query	**Relational Operator**
Is `A` greater than ‘B?	`A > B`
Is `A` greater than or equal to `B`?	`A >= B`
Is `A` less than `B`?	`A < B`
Is `A` less than or equal to `B`?	`A <= B`
Is `A` equal to (the same as) `B`?	`A == B`
Is `A` not (the same as) `B`?	`A ~= B`

Important Note: Beware of “==” and “=”. MATLAB treats them differently, and it will not always warn you of your mistake.

In Example 1, you can see some examples of these relational operators. They return logical values (`true` or `false`), which we will discuss in more detail later in this lesson.

Example 1

Given two variables a and b, conditionally check whether

a) `a` is less than `b`,

b) `a` is equal to `b`, or

c) `a` is not equal to `b`.

You may assume that a and b each store a value that is a real number.

Solution

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To demonstrate logical comparisons (conditions) in MATLAB
fprintf('To demonstrate logical comparisons (conditions) in MATLAB.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
%Initializing some random variables to compare
a = -2;
b = 5;
fprintf('The variables to compare are %g and %g.\n\n',a,b)

%------------------------- SOLUTION/OUTPUTS --------------------------
fprintf('OUTPUTS\n')
%Writing some conditions to evaluate
a < b   %Condition 1: check if 'a' is less than 'b'      (We expect TRUE)
a == b;  %Condition 2: check if 'a' is equal to 'b'       (We expect FALSE)
a ~= b;  %Condition 3: check if 'a' is NOT equal to 'b'   (We expect TRUE)

%Displaying results to a more readable format
cond1 = string(a < b)
cond2 = string(a == b)
cond3 = string(a ~= b)
```

In the last part of the solution, we convert the native output of a logical comparison into a more readable format using the MATLAB function `string()`. Observing the program outputs shown in Command Window reveals that a logical comparison like `a < b` has a messy output, which includes the tag `"logical"`. To convert this output to something more readable we use `string()`.

```
Command Window Output                                          Example 1
PURPOSE
To demonstrate logical comparisons (conditions) in MATLAB.

INPUTS
The variables to compare are -2 and 5.

OUTPUTS

ans =

  logical

   1


cond1 =

    "true"


cond2 =

    "false"


cond3 =

    "true"
```

What is Boolean logic?

Boolean values of 1 and 0, or `true` and `false`, respectively, represent a new data type in MATLAB called the `logical` data type. These values are binary (meaning they only have the two possibilities) and will act as such in all cases.

Conditional clauses (expressions that evaluate as true or false like `4 < 5` or `0 == 1`) can be stacked together with conditional-linking operators. That is, we can combine these conditional statements. We will cover the two most common condition-linking operators: `AND` and `OR`. Just like we use the conjunctions "and" and "or" in speech/language to join independent clauses, we *must* use them to join two or more conditions together in conditional expressions. Note that in Example 2 the conditional-linking operator is `AND` (represented by "`&&`") and `OR` (represented by "`||`").

- `AND` (&&): Both `condA && condB` must be true for the overall condition to be true.
- `OR` (`||`):Either `condA || condB` can be true for the overall condition to be true.

When using the `&&` comparison, all logic tests joined by the `&` must be true for the body of an if-statement to execute. For example, `3>2 && 7>8` would not execute the body of an if-statement. However, when the `||` comparison is used, only one of the joined tests must be true to execute the body. For instance, `3>2 || 7>8` would execute the body of the if-statement. Finally, `(3>2 && 7>8) || 1<=2` would evaluate as true since 1<=2 is true! This is an example of linking Boolean operators together, which is perfectly valid. As a side note, you can use `logical()` to convert numeric values to the logical data type in MATLAB. This might be especially useful when converting a matrix with numerical values to logical values.

Example 2

Given the variables, `a = 6` and `b = 3.4`, conditionally check whether a is greater than 1 and less than 5 and whether b is greater than 10 or equal to 3.4.

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To use Boolean logic
fprintf('To use Boolean logic.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining variables to compare
a = 6;
b = 3.4;
fprintf('The variables to compare are %g and %g.\n\n',a,b)

%------------------------- SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
a > 1 && a < 5  %True if 'a' is greater than 1 AND less than 5  (We expect FALSE)

b > 10 || b == 3.4  %True if 'b' is greater than 10 OR equal to 3.4  (We expect TRUE)
```

Command Window Output Example 2

```
PURPOSE
To use Boolean logic.

INPUTS
The variables to compare are 6 and 3.4.
```

Command Window Output (continued) Example 2

```
OUTPUTS

ans =

  logical

   0


ans =

  logical

   1
```

Can different data types be identified in MATLAB?

As you have likely experienced by now, data types must be handled with care. As a result, it can be useful to conditionally check the data type of a variable. MATLAB has handy functions for just such a purpose that return a Boolean value, which has a logical data type, of course. (We covered these previously in Lesson 2.5 (Data Types), and include them again here for clarity.)

These are called the data type identification functions, and some examples include testing whether a number is real or imaginary with `isreal()` or whether the value of a variable is a character data type with `ischar()`.

Example 3

Check whether a variable is a `char` data type or not. Output the class (data type) of the variable to the Command Window.

Solution

MATLAB Code example3.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To check whether a variable is a char data type or not
fprintf('To check whether a variable is a char data type or not.\n\n')
```

MATLAB Code (continued) example3.m

```
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining variables to test.
var1 = 'This is a string.';
var2 = 4;
fprintf('The variables to compare are “%s” and “%g”.\n\n',var1,var2)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
fprintf('The class of this variable is %s.\n',class(var1))
fprintf('Therefore, ischar(var1) = %s.\n\n',string(ischar(var1)))

fprintf('The class of this variable is %s.\n',class(var2))
fprintf('Therefore, ischar(var2) = %s.\n',string(ischar(var2)))
```

Command Window Output Example 3

```
PURPOSE
To check whether a variable is a char data type or not.

INPUTS
The variables to compare are “This is a string.” and “4”.

OUTPUTS
The class of this variable is char.
Therefore, ischar(var1) = true.

The class of this variable is double.
Therefore, ischar(var2) = false.
```

How can I round numbers in MATLAB?

Rounding functions can be very useful in writing effective conditions as we will demonstrate in the following lessons. First, though, we need to know the different rounding functions and how they work. Below is a list of the three most common rounding functions in MATLAB and what they do. You can see each of these functions implemented in MATLAB in Example 4.

- `round()`: returns the nearest integer (“normal” rounding)
  - Example: `round(1.5) = 2`
  - Example: `round(1.1) = 1`
- `ceil()`: returns the smallest integer that is greater than or equal to the number
  - Example: `ceil(1.1) = 2`
  - Example: `ceil(1.7) = 2`
- `floor()`: returns the greatest integer that is less than or equal to the number

  – Example: `floor(1.3) = 1`
  – Example: `floor(1.9) = 1`

Example 4

Show an example of how the MATLAB functions `round()`, `ceil()`, and `floor()` each round numbers.

Solution

MATLAB Code example4.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To demonstrate how rounding functions work in MATLAB
fprintf('To demonstrate how rounding functions work in MATLAB.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining variables to round
a = 8.5;
b = 8.1;
c = 8.9;
fprintf('The numbers to round are %g, %g, and %g\n\n',a,b,c)

%------------------------- SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
fprintf('round(%g) = %g\n',a,round(a))
fprintf('ceil(%g)  = %g\n',b,ceil(b))
fprintf('floor(%g) = %g\n',c,floor(c))
```

Command Window Output Example 4

```
PURPOSE
To demonstrate how rounding functions work in MATLAB.

INPUTS
The numbers to round are 8.5, 8.1, and 8.9

OUTPUTS
round(8.5) = 9
ceil(8.1)  = 9
floor(8.9) = 8
```

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Boolean AND operator	`&&`	`a && b`
Boolean OR operator	`||`	`a || b`
Round a number to the nearest integer	`round()`	`round(a)`
Round a number up to the nearest integer	`ceil()`	`ceil(a)`
Round a number down to the nearest integer	`floor()`	`floor(a)`
Check if a variable is a char data type or not	`ischar()`	`ischar(a)`
Check if a variable is a real number or not	`isreal()`	`isreal(a)`
Determine if A is greater than B	`>`	`A > B`
Determine if A is greater than or equal to B	`>=`	`A >= B`
Determine if A is less than B?	`<`	`A < B`
Determine if A is less than or equal to B	`<=`	`A <= B`
Determine if A is equal to (the same as) B	`==`	`A == B`
Determine if A is not (the same as) B	`~=`	`A ~= B`

Multiple Choice Quiz

(1). The `~=` operator stands for

(a) approximately equal to

(b) equal to

(c) greater than or equal to

(d) not equal to

(2). `sin(pi)==0` gives false as output because

(a) `sin(pi)=1`

(b) `sin(pi)=-1`

(c) `sin(pi)` is not defined

(d) `sin(pi)` gives a value other than zero in MATLAB

(3). The operator `||` stands for

(a) and

(b) or

(c) not

(d) not equal to

(4). The operator `&&` stands for

(a) and

(b) or

(c) not

(d) not equal to

(5). What is the Command Window output of the following program?

```
clc
clear
a = ceil(10.1)*round(4.1)*floor(1.5)
```

(a) `a = 60.6`

(b) `a = 44`

(c) `a = 60`

(d) `a = 66`

Problem Set

(1). Write a condition that evaluates as `true` when the given variable `length` is greater than 1.5. Test your condition using `length = 1` and then using `length = 3`.

(2). Write a condition that evaluates as `false` whenever the given variable `age` is less than 21. Test your condition using `age = 6` and then using `age = 30`.

(3). Write a condition that evaluates as `false` when `base` is equal to 5. Test your condition using a) `base = 0.2` and b) `base = 5`.

(4). Write a set of conditions that evaluates as `true` when the rounded value of the given variable `num` is greater than 16 and less than or equal to 21. Test your condition using a) `num = -8` and b) `num = 17`.

(5). Write a set of conditions that evaluates as `true` when the given variable `flag1` is equal to 2 or 3. Test your condition using a) `flag1 = 2` and b) `flag1 = 0`.

(6). An instructor wants to round up students' grades to the next integer. Write a program that takes students' grades as an input and returns the integer grades as an output. Hint: you will need to use a vector as the input/output.

Module 5: CONDITIONAL STATEMENTS

Lesson 5.2 – Conditional Statements: if and if-else

Learning Objectives

After reading this lesson, you should be able to:

- *construct logical expressions,*
- *use if statements to make conditional checks,*
- *use an if-else statement,*
- *make conditional checks if the statement is true or false,*
- *program conditional statements with Boolean logic and multiple expressions.*

So far in this text, we have considered only the basic control structure of a sequence. Sequence structure simply implies that statements are executed from the beginning to the end in a sequence. In this lesson, we introduce you to the control structure of conditions. This means that a programmer may want only a certain body of statements executed if a certain condition is true, and some other body of statements to be executed if that condition is false.

What is a conditional statement?

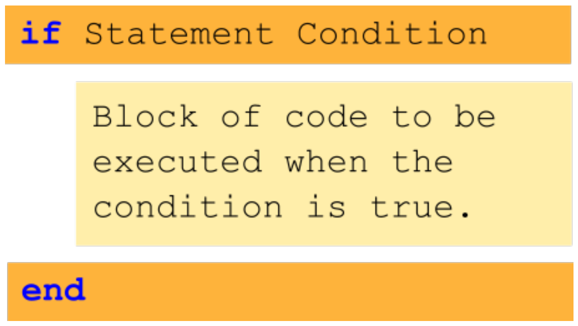
The conditional statement is one of the fundamental programming concepts. The conditional statement runs/executes a block of code if its condition is true. If its condition is false, that block of code will not execute. This is different than the conditional expressions (Boolean logic) we learned in Lesson 5.1. Conditional statements, when true, execute a block of code they contain (see Figure 1), while conditional expressions describe what makes them true (see Example 1).

For example, we might want to display a message that says whether a given number is above 10 (`x > 10`) or below 10 (`x < 10`). Obviously, the number cannot be both above *and* below 10, so we would need to display these messages conditionally.

There are two general types of conditional statements in MATLAB: `if` and `switch` statements. We will cover only the `if` statement since it is the most popular and versatile, but a very similar logic, applies to `switch` statements if you are curious. The `if` statement can be extended with the conditional clauses of `else` and `elseif`, which we will cover in this and the following lesson (Lesson 5.2).

What is the if statement?

An `if` statement is the simplest complete conditional statement. It needs only one condition, one line of code in the body, and an "`end`" to be complete. The logic test or expression is a condition which the `if` statement checks. The body is code that you want MATLAB to execute if the condition(s) is/are true. The body is defined by the lines of code between the `if` and `end` statements in the `if-end` conditional statement.



```
if Statement Condition
    Block of code to be
    executed when the
    condition is true.
end
```

Figure 1: Demonstrates the concept behind conditionally executing a block of code.

Once the `if-end` statement has reached the `end`, the rest of program will continue from the lines of code below the `end` of the `if-end` statement. In its simplest form, the `if-end` conditional statement has four main components. These are

1. the statement `if`
2. the logic test(s),
3. the body of the statement, and
4. the ending statement, `end`.

In Example 1, we use an `if` statement with a conditional expression. Here, we use one of the rounding functions we covered in Lesson 5.1 to design a conditional expression that fits our problem. Note that in Example 1 the `disp()` function will not be executed if the number is not an integer.

Example 1

Given a real number `num`, write a MATLAB program that displays, “The number is nonnegative”, if the input number is nonnegative (a number is nonnegative if it is zero or positive), or “The number is negative” if the input number is negative. Run and test your program twice, using values of `-4.5` and `6.3`. Use only `if-end` statements.

Solution

The program uses two `if-end` statements, the first checks if the input number is positive, and the second checks if the input number is negative. When writing the program for this example, one must be careful so as not to create a situation where both statements are true. Notice the difference in logic tests between the two statements.

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To check if a number is negative or nonnegative
fprintf('To check if a number is negative or nonnegative.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining number to check. Assume it is a number (numeric data type).
num = -4.5;
fprintf('The number to check is %g.\n\n',num)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
%Check if num is greater than or equal to zero
if num >= 0
    disp('The number is nonnegative.')
end
%Check if num is less than zero
if num < 0
    disp('The number is negative.')
end
```

The Command Window output for Example 1 when `num = -4.5` is shown next.

Command Window Output Example 1

```
PURPOSE
To check if a number is negative or nonnegative.

INPUTS
The number to check is -4.5.

OUTPUTS
The number is negative.
```

The Command Window output for Example 1 when `num = 6.3` is shown next.

Command Window Output Example 1

```
PURPOSE
To check if a number is negative or nonnegative.

INPUTS
The number to check is 6.3.

OUTPUTS
The number is nonnegative.
```

What is the if-else statement?

The `else` conditional clause introduces a new component to a conditional statement. Note, `else` is not a stand-alone component. If used, it must always follow an `if` statement. Its function is just as it reads: *if* this condition is true, run this block of code; *else*, run this other block of code. Therefore, for the `else` block of code to run, the `if` condition must be false. This concept can be seen in Figures 2 and 3.

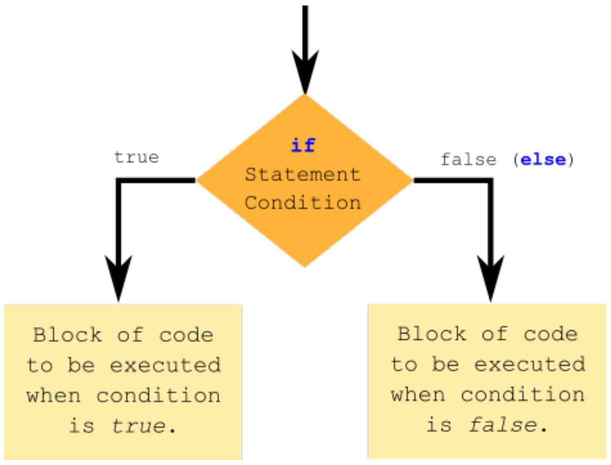

true
if
Statement
Condition
false (else)
Block of code to be executed when condition is *true*.
Block of code to be executed when condition is *false*.

Figure 2: Flowchart of an if-else conditional statement.

Figure 2 shows a more conceptual representation of an if-else conditional statement, while Figure 3 is more of a coding representation (similar to pseudocode). We will go into more detail about flowcharts and pseudocode in Module 6.

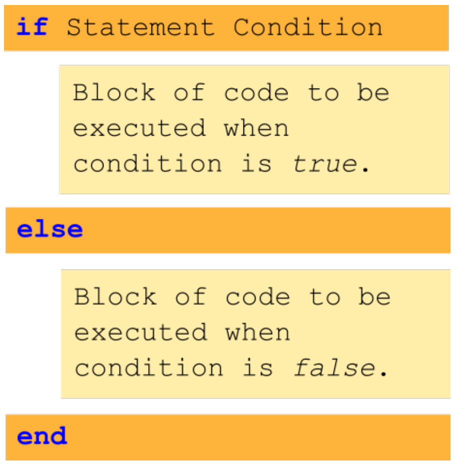

if Statement Condition
Block of code to be executed when condition is *true*.
else
Block of code to be executed when condition is *false*.
end

Figure 3: Shows how each conditional clause (`if` and `else`) each has its own block of code.

***Important Notes*:**

- `else` is not a stand-alone clause: it must always follow `if`.
- For the `else` block of code to run, the `if` clause must be false.
- There is only one `end` per conditional statement as seen in Figure 3. One `end` per `if`: NOT `if-end-else-end`.

In Example 2, we add the `else` or "otherwise" case, which lets us give the user more information in case the `if` statement is false. For instance, in Example 1, all we could do was to output a message when the `if` statement was true. Now, we can execute a block of code when the given number is a nonnegative integer and execute another (different) block of code when the number is negative.

Example 2

Given a real number `num`, write a MATLAB program that displays, "The number is nonnegative", if the input number is nonnegative (a number is nonnegative if it is zero or positive), or "The number is negative" if the input number is negative. Run and test your program twice, using values of `-4.5` and `6.3`. Unlike Example 1, where you were only allowed to use if-end statements, use only the if-else-end statement(s).

Solution

Although the output is the same as that of Example 1, this m-file is slightly shorter. This is because MATLAB only needs to conduct one logic test here as opposed to conducting two logic tests in Example 1.

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To check if a number is negative or nonnegative
fprintf('To check if a number is negative or nonnegative.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining number to check. Assume it is a number (numeric data type).
num = -4.5;
fprintf('The number to check is %g.\n\n',num)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
%Check if num is greater than or equal to zero; else, it is negative.
if num >= 0
    disp('The number is nonnegative.')
else
    disp('The number is negative.')
end
```

As you can see in Example 2, only the body of the `if` statement is executed. Once a conditional clause evaluates as true, the block contained is run and then the whole conditional clause is exited. This means `else` is not considered nor is its code block executed. This is another way of saying the `if` condition(s) must be false for `else` to run.

The Command Window output for Example 2 when `num = -4.5` is shown next.

Command Window Output Example 2

```
PURPOSE
To check if a number is negative or nonnegative.

INPUTS
The number to check is -4.5.

OUTPUTS
The number is negative.
```

The Command Window output for Example 2 when `num = 6.3` is shown next.

Command Window Output Example 2

```
PURPOSE
To check if a number is negative or nonnegative.

INPUTS
The number to check is 6.3.

OUTPUTS
The number is nonnegative.
```

✓ ***Important Note:*** Because a condition cannot be simultaneously true and false, an `if-else` statement, only the `if` *or* the `else` block runs: *never both*.

Can I use multiple conditions in a single expression?

We saw how to join two conditional expressions together in Lesson 5.1 with Boolean logic using `AND` (`&&`) and `OR` (`||`). Now we will look at how to use multiple conditional expressions in a single `if` statement.

In Example 3, we use multiple conditional expressions in an `if` statement, which allows us to remove the assumption that the variable holds a number. Try it yourself with a char data type and see what happens!

Example 3

Write a program that checks if a variable is a numeric data type and an integer. Display the result to the Command Window.

Solution

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To check if a variable is a number and an integer
fprintf('To check if a variable is a number and an integer.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
var = 10;  %Defining number to check.
fprintf('The variable to check is %g.\n\n',var)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
%Check if the variable is a number AND an integer
if isnumeric(var) && ceil(var) == var
    disp('This variable is a number and an integer.')
else    %Note, we cannot tell which of the conditions failed here.
    disp('This variable is not a number and/or not an integer.')
end
```

```
Command Window Output                                    Example 3
PURPOSE
To check if a variable is a number and an integer.

INPUTS
The variable to check is 10.

OUTPUTS
This variable is a number and an integer.
```

In Example 3, we use two compounded conditional statements joined by an AND (`&&`) operator. In Example 4, we provide a usage case of the Boolean operator OR (`||`).

Example 4

In 1998, the federal government introduced the body mass index (BMI) to determine healthy weights. Body mass index is calculated as 703 times the weight in pounds divided by the square of the height in inches of the individual. The obtained number is then *rounded off* to the nearest whole number. The criterion for a healthy weight is given as follows.

- BMI < 19 - Unhealthy weight
- $19 \leq BMI \leq 25$ - Healthy weight
- BMI > 25 - Unhealthy weight

Write a MATLAB program that outputs, based on the above criterion, if an individual has a healthy or unhealthy weight. The program inputs are the person's weight in pounds and height in inches.

Solution

A person's BMI is calculated by the formula:

$$\text{BMI} = \frac{\text{weight (lbs)}}{[\text{height (in)}]^2} \times 703$$

The steps in the algorithm are:

1. Enter the person's weight in lbs and height in inches.
2. Calculate BMI using, $\text{BMI} = \dfrac{\text{weight (lbs)}}{[\text{height (in)}]^2} \times 703$
3. If BMI not in the range of 19 and 25, then the person has an unhealthy weight, else the weight is healthy.

MATLAB Code example4.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To calculate BMI
fprintf('To calculate BMI.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Getting necessary variables from the user
weight = input('Your weight (lbs):');
height = input('Your height (in):');

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('\nOUTPUTS\n')
BMI = (weight/height^2)*703;  %Calculating BMI

%Displaying results to user
fprintf('Your BMI is %g.\n',BMI)

if BMI > 25 || BMI < 19
    disp('This is an unhealthy BMI.')
else
    disp('This is a healthy BMI.')
end
```

Command Window Output Example 4

```
PURPOSE
To calculate BMI.

INPUTS
Your weight (lbs):155
Your height (in):68

OUTPUTS
Your BMI is 23.5651.
This is a healthy BMI.
```

A Note on Writing Good Conditional Statements

As we have previously discussed in Lesson 2.1, mixing double negatives and Boolean values is a bad practice and adds unnecessary complexity in the long run (“ain’t no good”). If you are using variable names inside your conditions, make sure you follow this rule. You can see a detailed explanation in that lesson if you need a review.

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Conditionally execute a single block of code	`if`	`if 3<5; disp('T'); end`
Conditionally execute code for both the true and false cases of the condition	`else`	`if 3>5; disp('T'); else disp('F'); end`
Use the Boolean AND operator	`&&`	`a && b`
Use the Boolean OR operator	`||`	`a || b`

Multiple Choice Quiz

(1). What is the Command Window output of the following program?

```
clc
clear
A = 10;
if A >= 10
      B = 5;
else
      B = 6;
end
B
```

(a) `5`

(b) `6`

(c) `10`

(d) `11`

(2). What is the Command Window output of the following program?

```
clc
clear
A = 10;
if A >= 10
      B = 5;
else
      B = 6;
end
A
```

(a) `5`

(b) `6`

(c) `10`

(d) `11`

(3). What is the Command Window output of the following program?

```
clc
clear
B = 79;
G = 2;
if B > 50
      G = 1;
end
if B > 60
      G = 3;
end
G
```

(a) `1`

(b) `2`

(c) `3`

(d) `79`

(4). What is the Command Window output of the following program?

```
clc
clear
A = 6;
B = 3;
if A > 5 || B < 1
      A = 20;
end
A
```

(a) `3`

(b) `5`

(c) `6`

(d) `20`

(5). What is the Command Window output of the following program?

```
clc
clear
A = 6;
B = 3;
if A > 5 && B < 1;
      A = 20;
end
A
```

(a) `3`

(b) `5`

(c) `6`

(d) `20`

Problem Set

(1). Using only conditional statements, write a MATLAB program that determines if any input number, `inputNum`, is an integer or a decimal number. The program output is "The input number is an integer" or "The input number is a decimal number".

(2). Using only conditional statements, write a MATLAB program that determines if an integer is even or odd. Display a message to the Command Window telling the user whether the integer is even or odd.

(3). The United States House of Representatives has 435 members. For a bill to pass the house, a simple majority (50% or more) of members have to vote in favor of the bill. Write a program given the variable yes that appropriately outputs either "The bill passed!" or "The bill did not pass!" to the Command Window.

(4). A student's grade is based on their score in four categories: homework, projects, tests, and final exam. The weight of each category and the student's scores are given in Table A.

Table A: Student grade information.

Category	**Weight**	**Example Scores**
Homework	10%	88
Projects	12%	95

Category	**Weight**	**Example Scores**
Tests	48%	76
Final exam	30%	91

Write a program that calculates the overall percentage grade of a student. Round up the overall percentage grade to the next integer. Determine if a student has passed or failed the course if the passing grade is 70% or higher.

Use the example case given in Table A to test your solution.

(5). A simply supported beam is loaded as shown in Figure A. Under the applied load, the beam will deflect vertically. This vertical deflection of the beam V will vary along the length of the beam from $x = 0$ to L and is given by

$$V = \begin{cases} \dfrac{Pb}{6EIL}\left[(-L^2 + b^2)x + x^3\right], 0 < x < a \\ \dfrac{Pb}{6EIL}\left[(-L^2 + b^2)x + x^3 - \dfrac{L}{b}(x-a)^3\right], a < x < L \end{cases}$$

where,

- x is the distance from the left end,
- P is the load,
- L is the length of the beam,
- a is the location where the load P is applied,
- E is Young's modulus of the beam material, and
- I is the second moment of area.

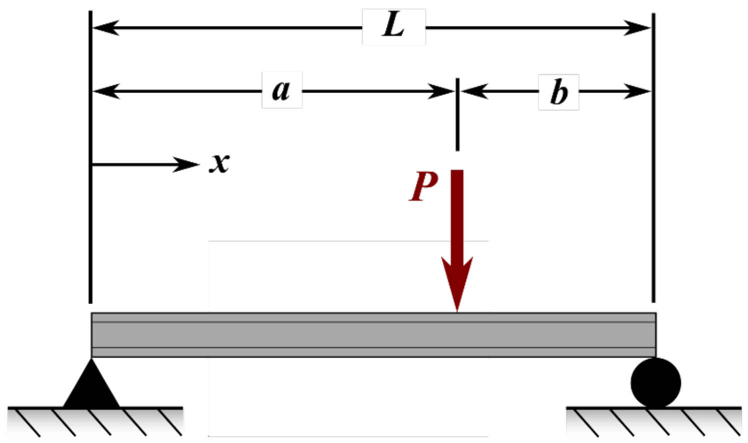


Figure A: Simply supported beam shown with applied load, P.

Write a MATLAB program that outputs the vertical deflection of the beam at a point of interest. Display all of the inputs and outputs by using the `fprintf()` function complete with explanation and reasonable format.

The program inputs, as entered by `input()` functions, are

1. distance from the left end to the point of interest, x,
2. length of the beam, L,
3. load, P,
4. the location where the load P is applied, a,
5. Young's modulus of the beam material, E, and
6. the second moment of area, I,

and output is

1. the calculated deflection, V

Run your program for the following two input sets. You will need to run the m-file twice: once for each of the two input sets.

1. $x = 2.50, \quad L = 5, \quad a = 3, \quad E = 30 \times 10^6, \quad I = 0.0256, \quad P = 30$
2. $x = 4.05, \quad L = 5, \quad a = 3, \quad E = 30 \times 10^6, \quad I = 0.0256, \quad P = 30$

Module 5: CONDITIONAL STATEMENTS

Lesson 5.3 – Conditional Statements: if-elseif

Learning Objectives

After reading this lesson, you should be able to:

- *use the if-elseif statement to address multiple conditional cases,*
- *identify when to use multiple if-end statements,*
- *program an if-elseif-else statement,*
- *identify the differences between elseif and else.*

What is the if-elseif statement?

An `elseif` conditional clause is the third and last type of conditional clause we cover. An `elseif` clause does essentially the same thing as an `if` statement since it has its own condition: *except* it will only be evaluated when its parent (and previous) `if` statement is false (see Example 1). The `elseif` clauses can also have multiple conditional expressions just like we discussed in Lesson 5.2 for `if` statements.

Example 1

Given a real number `num`, write a MATLAB program that displays, “The number is positive” if the input number is positive, or “The number is negative” if the input number is negative, or “The number is zero” if the input number is zero. Run and test your program three times, using the values of num of `-4.5`, `0`, and `6`.

Solution

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To determine if a number is positive, zero, or negative
fprintf('To determine if a number is positive, zero, or negative.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
num = -4.5;   %Define a variable as some number we would like to check
fprintf('The number to test is %g.\n\n',num)
```

MATLAB Code (continued) example1.m

```
%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
if num > 0       %The if statement is always checked first
    fprintf('The sign of the number is positive.\n')
elseif num == 0  %This only runs when the if statement is false
    fprintf('The number is zero.\n')
else             %This only runs when the if and elseif statements are false
    fprintf('The sign of the number is negative.\n')
end
```

Command Window Output Example 1

```
PURPOSE
To determine if a number is positive, zero, or negative.

INPUTS
The number to test is -4.5.

OUTPUTS
The sign of the number is negative.
```

Only one condition (`if` or `elseif`) will be true at once. It is also essential to note that the whole linked conditional statement is terminated by a single end.

Example 2 illustrates another use of the `if-elseif` statement. Here, we want to check for a certain string and conditionally run our block of code based on that. We use a special function `strcmp()` to avoid a “matrix dimension mismatch” error from MATLAB when comparing strings.

Example 2

Provide a greeting based on a username entered by a string. Display the result to the Command Window.

Solution

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To compare strings in a condition statement
fprintf('To compare strings in a condition statement.\n\n')
```

MATLAB Code (continued) example2.m

```
%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
userName = 'billy-bob';  %Defining the username to check
fprintf('The username is %s.\n\n',userName)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
if strcmp(userName,'mary-ann')      %The if clause is always checked first
    fprintf('Hey Mary-Ann! How the horses feelin''?\n')

elseif strcmp(userName,'billy-bob') %This clause is checked when the previous
                                    %    clause(s) are false
    fprintf('Hey Billy-Bob! You ready for the hike up the mountain?\n')
end
```

Command Window Output Example 2

```
PURPOSE
To compare strings in a condition statement.

INPUTS
The username is billy-bob.

OUTPUTS
Hey Billy-Bob! You ready for the hike up the mountain?
```

✓ ***Important Note:*** In an `if-elseif-else` statement, *only* the `if` *OR* the `elseif` *OR* the `else` runs.

Independent vs. Dependent Cases

In this context, a conditional case just means that we wish to consider a specific scenario, which is described by one or more individual conditions. Independent cases can all be true simultaneously (or independently). They do not depend on each other to be true or false (see Example 3). Independent cases should be implemented with separate `if` statements. For example, the width and height of a beam are independent conditions (neglecting application requirements).

Dependent cases cannot be true simultaneously and should be implemented with `elseif/else` clauses (see Example 4). Take Example 1, for instance. These are dependent cases because only one can be true for a given input. Another example is checking the value of the cross-sectional area of a beam. The area cannot be two values at once, so checking whether the area is X or Y should be implemented as dependent cases.

Example 3

Check whether the two independent variables x and y are within given bounds. The value of x should be less than 5, and that of y should be greater than 20. Display the result to the Command Window.

Solution

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To check the status of two independent conditions
fprintf('To check the status of two independent conditions.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
%Defining arbitrary variables for conditional statements
x = 3;
y = 22;
fprintf('The variables to check are %g and %g.\n\n',x,y)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
if x < 5
    disp('x is less than 5.')
end

if y > 20
    disp('y is greater than 20.')
end
```

Combining these if-end statements as an if-elseif statement (`if x<5` and `elseif y>20`) would only result in the x message displaying (with the current values of `x` and `y`). You should try this for yourself!

Command Window Output Example 3

```
PURPOSE
To check the status of two independent conditions.

INPUTS
The variables to check are 3 and 22.

OUTPUTS
x is less than 5.
y is greater than 20.
```

In Example 4, both conditions (cases) depend on `a`, so they are dependent. The variable `a` cannot be two values simultaneously.

Example 4

Check whether the variable a is less than five or not. Display the result to the Command Window.

Solution

MATLAB Code example4.m

```
clc
clear

%------------------------------ PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To check the status of two dependent cases
fprintf('To check the status of two dependent cases.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
a = 3;  %Defining arbitrary variable for conditional statement
fprintf('The value to check is %g.\n\n',a)

%------------------------- SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
if a < 5
    disp('a is less than five.')
elseif a >= 5
    disp('a is greater than or equal to five.')
end
```

Command Window Output Example 4

```
PURPOSE
To check the status of two dependent cases.

INPUTS
The value to check is 3.

OUTPUTS
a is less than five.
```

If you are still unsure about the differences between dependent and independent conditions, just take a step back when you are programming and think whether the two (or more) scenarios you are considering can be true simultaneously in the "real world". Can a = 2 AND a = 3? No! You are implementing "real world" scenarios when you program, so you should first know what is possible or not in the real world. Then write a program structure that implements it in your code.

What is the if-elseif-else statement?

We can put everything we have learned together with an `if-elseif-else` statement. As you can see in Example 5, `else` must *always* come last! That is, if all previous conditions are false, the code block under the else statement will be executed.

Example 5

Monitor the temperature of an oven. Write a program that determines whether the oven is still heating up, over the target temperature, or at the target temperature. The two necessary temperature values are given as follows. The last temperature reading from the temperature sensor of the oven is 400°F. The bake temperature set by the user is 425°F.

Solution

MATLAB Code example5.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To monitor the status of an oven's temperature
fprintf('To monitor the status of an oven''s temperature.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
targetTemp  = 425;  %The temperature we want to reach (set on the oven)
currentTemp = 400;  %This is a reading from the temp sensor inside the oven
fprintf('The target temperature is %gF.\n',targetTemp)
fprintf('The current temperature reading is %gF.\n\n',currentTemp)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
if currentTemp == targetTemp     %Check if oven is ready
    disp('Oven is ready!')
elseif currentTemp < targetTemp  %Check if oven needs to continue heating
    disp('Oven is heating up.')
else    %If none of the above clauses are true, the oven must be over temp
    disp('Oven is over the target temperature.')
end
```

```
Command Window Output                                    Example 5
PURPOSE
To monitor the status of an oven's temperature.

INPUTS
The target temperature is 425F.
The current temperature reading is 400F.

OUTPUTS
Oven is heating up.
```

Notice these are dependent cases (there is only one temperature value at a given time), so the cases are implemented with an `if-elseif-else` statement. Using an `if-elseif-elseif` statement would also yield a correctly functioning program, but it would be slightly less computationally efficient since there would be an extra condition to check in the second `elseif` clause.

What is the difference between the else and elseif conditional clauses?

An `elseif` has its own condition, whereas `else` has no condition and will always run given that all conditional clauses (`if` or `elseif`) before it are false. Therefore, you should only use an `else` when you want to address *all other possibilities*. Use `elseif` when you want to address specific cases only, and when you want to address multiple specific cases that are unique (i.e., the cases cannot be true at the same time: a $>$ 0 and a $<$ 0).

Important Note: `else` must *always* come last in a conditional statement.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Example Usage
Conditionally execute blocks of code in multiple exclusive areas	`elseif`	`if a<5; disp('case 1');` `elseif a>8; disp('case2');` `end`

Multiple Choice Quiz

(1). Which of the following statements is not true about the `elseif` clause?

(a) `elseif` must always be used in conjunction with an `if` statement.

(b) `elseif` must have its own condition.

(c) An `elseif` clause is not always executed.

(d) One may only use up to three `elseif` clauses in a single `if-end` statement.

(2). What will the Command Window output of the following program be?

```
clc
clear
A = 4.1;
if A >= 3
     B = 13;
elseif A = 0
     B = 4.2
end
A
```

(a) `A = 4.1`

(b) `B = 13`

(c) `Incorrect use of '=' operator.`

(d) `A = 3`

(3). What is the Command Window output of the following program?

```
clc
clear
A = 4.1;
if A >= 3
     B = 13;
elseif A = 0
     B = 4.2
end
A
```

(a) `Case A`

(b) `Case B`

(c) `Case C`

(d) `Case D`

(4). What is the Command Window output of the following program?

```
clc
clear
a = 11;
b = 6;
if a < 10 || b ~= 10
     disp('Option 1')
elseif a == 7
     disp('Option 2')
else
     disp('None')
end
```

(a) `Option 1`

(b) `Option 2`

(c) `None`

(d) The program displays nothing in the Command Window.

(5). Fill in the blank with one of the choices to make this code output `b = 6`.

```
clc
clear
a = 11.5;
b = 0.2;
if ____________

    b = 6;
elseif a == 4.5
    b = 4;
else
    b = 2;
end
b
```

(a) `a==b`

(b) `a<=b`

(c) `a~=b`

(d) `a<b`

Problem Set

(1). Given an integer, write a MATLAB code for displaying whether the integer is a) positive (greater than zero), b) zero, c) or negative (less than zero). In

Example 1 the solution is given using `elseif`, but here you are asked to use three separate `if-end` statements to complete the solution.

(2). A student's grade is based on their score in four categories: homework, projects, tests, and final exam. The weight of each category and the student's scores are given in Table A below.

Table A: Student grade information.

Category	**Weight**	**Example Scores**
Homework	10%	88
Projects	12%	95
Tests	48%	76
Final exam	30%	91

The letter grades are given by

- $90 \leq \text{A} \leq 100$,
- $80 \leq \text{B} < 90$,
- $70 \leq \text{C} < 80$,
- $60 \leq \text{D} < 70$, and
- $\text{F} < 60$

Write a program that calculates the overall percentage and letter grade for any student in the class and displays these in the Command Window. Use the given example as your test case.

(3). Your friend is having a hard time keeping track of their weekly schedule, which is as follows:

- Monday – Class at 7:30 AM and work at 5:00 PM
- Tuesday – Class at 2:00 PM and 4:30 PM
- Wednesday – Weekly group progress meeting at 11:30 AM and no class
- Thursday – Same as Tuesday
- Friday – Same as Monday

Your friend uses MATLAB every day, and therefore you told him you would write a program to keep track of his daily events. Using your knowledge of conditional case-structure, write a program where the output is a string of characters containing the events of a single day (not the entire week of events). The program input will be an integer which corresponds to a day of the week, where 1 is for Monday, 2 is for Tuesday, etc.

Provide an error statement if a number other than `1`,`2`,`3`,`4` or `5` is entered. Test your program using an input of 4 (corresponding to Thursday).

(4). So, you want my phone number and need to know my BMI? How shallow can you be? In 1998, the federal government introduced the body mass index (BMI) to determine an ideal weight based on a person's height. Body mass index is calculated as 703 times the weight in pounds divided by the square of the height in inches, the obtained number is then **rounded off** to the nearest whole number (Hint: 23.5 will be rounded to 24; 23.1 will be rounded to 23; 23.52 will be rounded to 24). The criterion for the weight category is given as follows:

- BMI $<$ 19 - Underweight
- 19 $\leq$ BMI $\leq$ 25 - Healthy weight
- 25 $<$ BMI $\leq$ 30 - Overweight
- BMI $>$ 30 - Obese

Develop a MATLAB program that outputs an integer based on the person's input weight, `w`, and height, `h`. The integer 0, 1, 2, or 3, is the output, depending on if a person is underweight, healthy weight, overweight, or obese, respectively.

Develop a program where based on a person's input weight, `w`, and height, `h`, it outputs a description of their health state and a target healthy weight (if needed). The description to the health state output is, "Underweight", "Healthy weight", "Overweight", or, "Obese", and the target weight is to be rounded to the whole integer. Use the `fprintf()` function to describe all inputs and outputs.

(5). United States citizens pay federal income tax, social security tax, and Medicare tax on their wages.

Federal Income Tax: According to the Internal Revenue Service, a single-status U.S. citizen will pay 2018 federal income taxes according to the following chart*.

10% on income between \$0 and \$9,525

12% on the income between \$9,526 and \$38,700; plus \$952.50

22% on the income between \$38,701 and \$82,500; plus \$4,453.50

24% on the income between \$82,501 and \$157,500; plus \$14,089.50

32% on the income between \$157,501 and \$200,000; plus \$32,089.50

35% on the income between \$200,001 and \$500,000; plus \$45,689.50

37% on the income over \$500,001; plus \$150,689.50

Social Security tax: In 2018, this is 6.2% on earnings up to \$128,400**

Medicare tax: In 2018, this is 1.45% on all earnings.** An additional 0.9% applies to individuals who earn over \$200,000.***

Complete parts (a) and (b):

a. Construct a pseudo code for a MATLAB program to find a person's total tax liability based on their income. Assume all income is taxable.

b. Assume all income is taxable and write a program that inputs

1. the person's income, `income`,

and then outputs the owed

1. federal tax, `fedTax`,
2. social security tax, `ssTax`,
3. medicare tax, `medTax`.
4. total taxes, `totalTax`.

Display all of the inputs and outputs by using the `fprintf()` function, complete with an explanation in a reasonable format.

Run the program with incomes of \$32,000, \$85,000 and \$269,000.

References:

* Kelly Phillips Erb, "New: IRS Announces 2018 Tax Rates, Standard Deductions, Exemption Amounts And More" <https://www.forbes.com/sites/kellyphillipserb/2018/03/07/new-irs-announces-2018-tax-rates-standard-deductions-exemption-amounts-and-more/#28e98d0d3133>

**U.S. Government, "Update 2019 – Social Security" <https://www.ssa.gov/pubs/EN-05-10003.pdf>

*** U.S. Government, “Questions and Answers for the Additional Medicare Tax”
<https://www.irs.gov/businesses/small-businesses-self-employed/questions-and-answers-for-the-additional-medicare-tax>

Module 6: PROGRAM DESIGN AND COMMUNICATION

Lesson 6.1 – Flowcharts

Learning Objectives

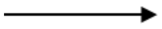
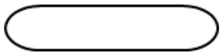
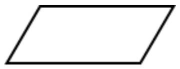
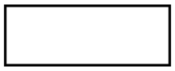
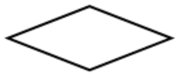
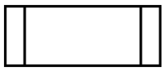
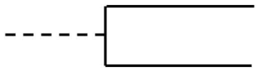
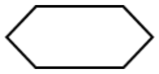
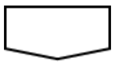
After reading this lesson, you should be able to:

- *identify flowchart symbols,*
- *construct a flowchart for a program,*
- *read a flowchart.*

What is a flowchart?

A flowchart is the combination of geometric symbols connected with arrows (called flowlines) that represent steps in a process. Each geometric symbol has a meaning (for example, a rectangle represents a process), and is filled with necessary information (for instance, a formula in a process). Flowcharts are commonly used as a tool to help layout and plan large-scale processes and programs. It is also very common to find flowcharts as part of engineering plans and corporate meetings. Table 1 shows commonly used flowchart symbols with their meaning.

Table 1: Common flowchart symbols and meanings.

Symbol	**Title**	**Meaning**
	Flowline	connect symbols and indicate the flow of logic.
	Terminal	To represent the beginning or end of a task.
	Input/Output	For input and output operations.
	Processing	For arithmetic and data manipulation operations.
	Decision	For the logic of comparison operations.
	Connector	Used to join different flowlines.
	Predefined Process	To represent a group of statements that performs one processing task.
	Annotation	To provide additional information about another flowchart symbol.
	Loop	Used for loops/repetition
	Offpage Connector	To indicate that the flowchart continues to a second page.

Example 1

Hero's formula for calculating the area of a triangle with the length of the three sides as a, b, c is given by,

$A = \sqrt{s(s-a)(s-b)(s-c)}$,

where

s is the semi-perimeter of the triangle, that is,

$s = \frac{a+b+c}{2}$.

The perimeter of the triangle is given by

$$P = a + b + c$$

Construct a flow chart for calculating the perimeter and area of a triangle.

Solution

(*Flowchart is shown on the next page.*)

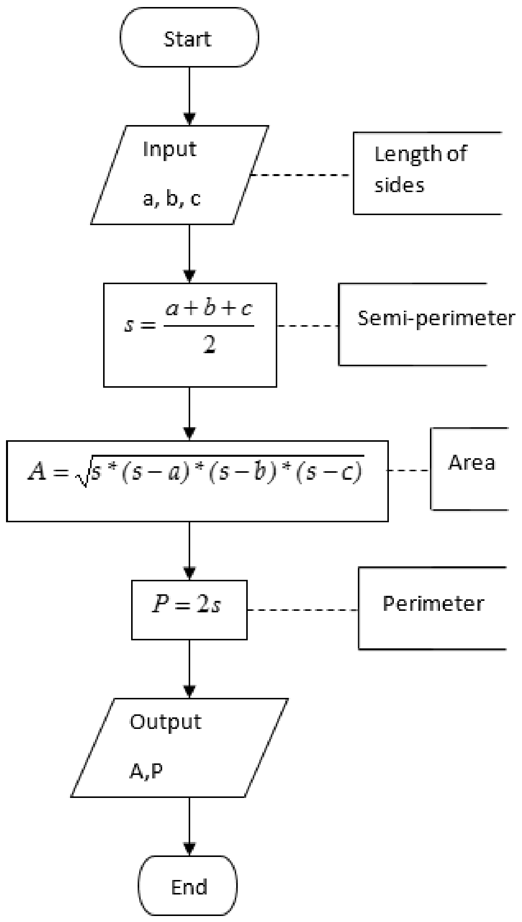


Figure 1: Flow chart for calculating the area and perimeter for a triangle.

Example 2

So, you want my phone number and need to know my BMI? How shallow can you be? In 1998, the federal government introduced the body mass index (BMI) to determine healthy weights.

Body mass index is calculated as 703 times the weight in pounds divided by the square of the height in inches. The obtained number is then **rounded off** to the nearest whole number (Hint: 23.5 will be rounded to 24; 23.1 will be rounded to 23; 23.52 will be rounded to 24). The criteria for a healthy weight are given as follows.

Table 2: Range of BMI values.

Range of BMI	**Meaning of Range**
$BMI < 19$	Unhealthy weight
$19 \leq BMI \leq 25$	Healthy weight
$BMI > 25$	Unhealthy weight

Construct a flow chart for the above example that will determine whether a person has a healthy or unhealthy weight, based on a person's weight and height.

Solution

A person's BMI is calculated by the formula

$BMI = \frac{weight(lbs)}{[height(in)]^2} \times 703.$

The steps in the algorithm are:

1. Enter the person's weight in lbs and height in inches.

2. Calculate BMI using, $BMI = \frac{weight(lbs)}{[height(in)]^2} \times 703.$

3. If BMI is not in the range of 19 and 25, then the person has an unhealthy weight, else the weight is healthy.

The flowchart for this problem is shown in Figure 2.

(*Flowchart is shown on next page*)

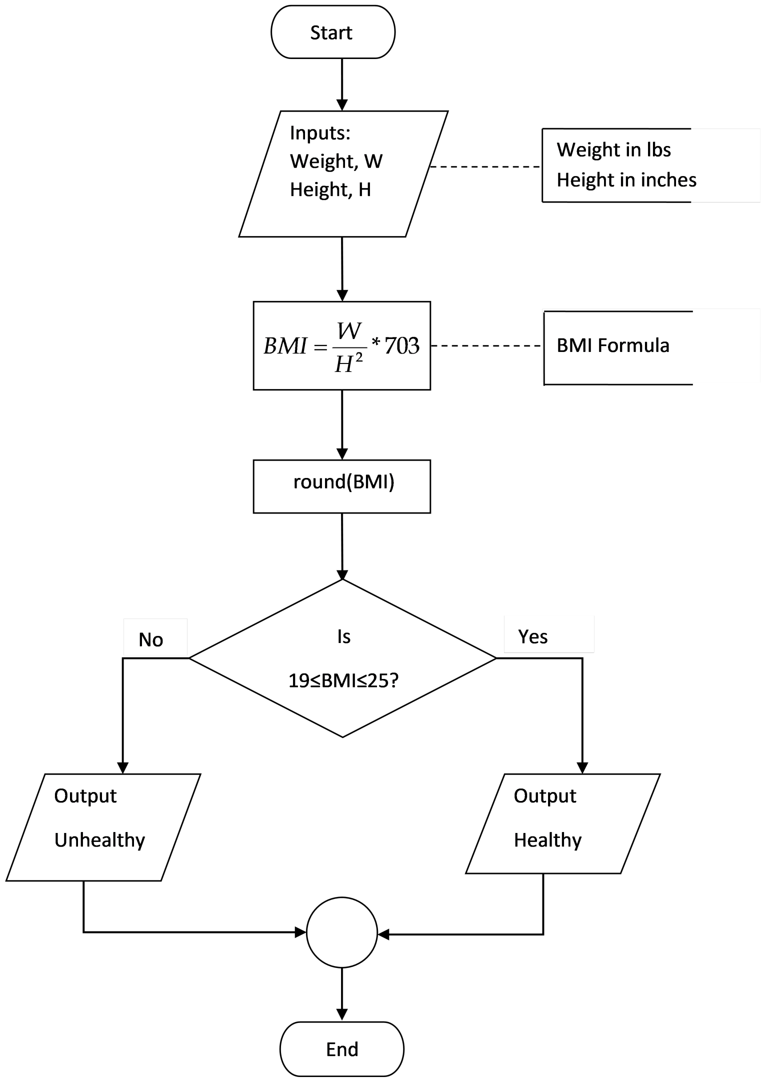


Figure 2: Flow chart for calculating a person's BMI.

Example 3

The function e^x can be calculated by using the following infinite Maclaurin series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Make a flowchart for finding e^x using the first n terms of the Maclaurin series.

Solution

This solution uses loops to develop the flowchart. Although loops have not been covered in this book yet (Module 8), loops are simply a repetitive task, starting and ending at some value. An example of using a loop is a summation series.

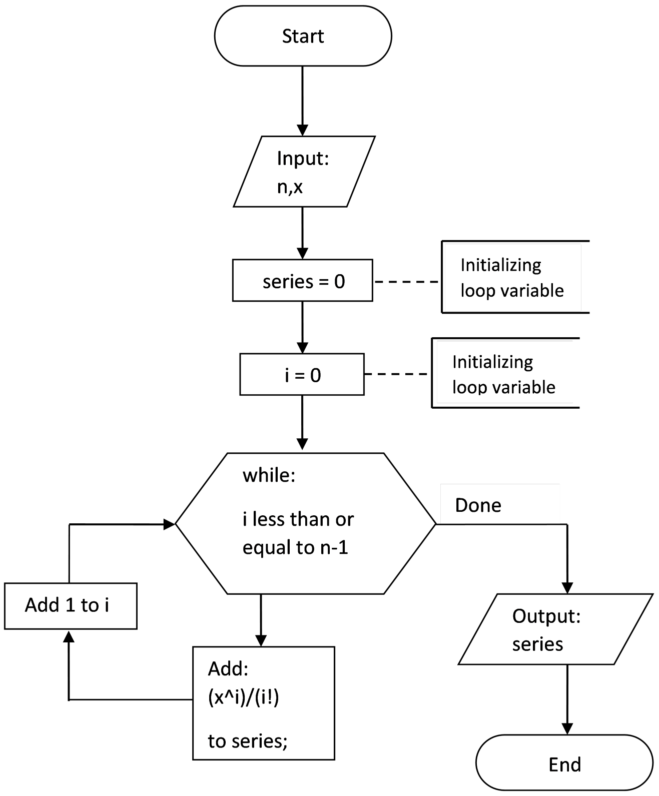
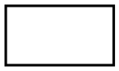
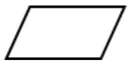

Start
Input: n,x
series = 0
Initializing loop variable
i = 0
Initializing loop variable
while: i less than or equal to n-1
Done
Add 1 to i
Add: (x^i)/(i!) to series;
Output: series
End

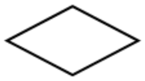
Figure 3: Flowchart for calculating e^x using the Maclaurin Series.

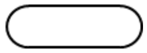
Multiple Choice Quiz

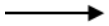
(1). The flowchart symbol for a process is

(a) 

(b) 

(c) 

(d) 

(2). The  symbol in a flowchart represents

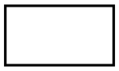
(a) the direction that information travels in a process.

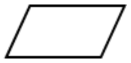
(b) the direction of least resistance in a process.

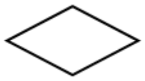
(c) the reverse direction of data flow.

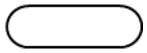
(d) a visual distraction.

(3). The flowchart symbol for a decision is

(a) 

(b) 

(c) 

(d) 

(4). The flowchart symbol, , is used to

(a) specify a process.

(b) start or end a flowchart.

(c) link two or more flowlines.

(d) add a new input or output.

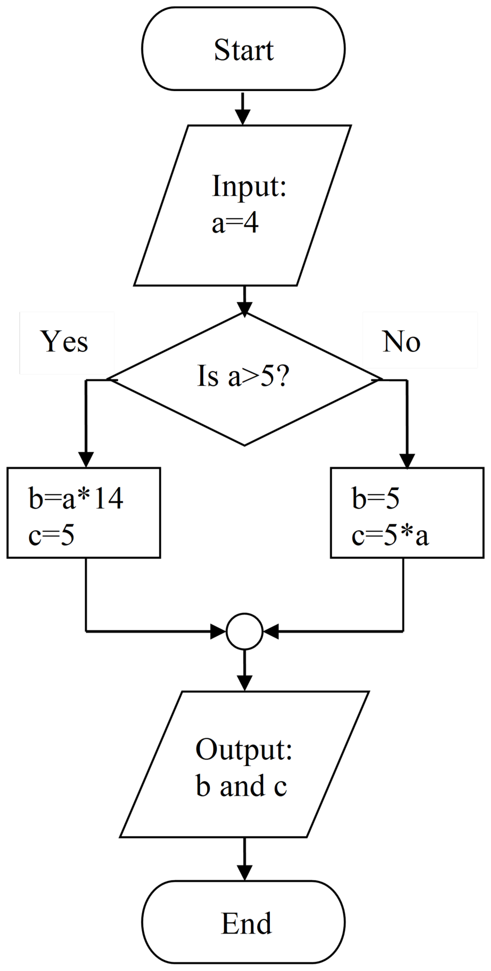
(5). Given the following flowchart, what would be the output

(a) a = 4, c = 5

(b) b = 56, c = 5

(c) b = 5, c = 20

(d) b = 20, c = 5



Problem Set

(1). Construct a flowchart for the process of finding the area of a trapezoid, given the length of the two parallel sides and the perpendicular distance between the parallel sides.

(2). Construct a flowchart for determining if an object will fit in a three-dimensional box. To do this, compare the dimensions of the object to the dimensions of the box. Both sets of dimensions should be inputs to the program.

(3). Construct a flowchart for determining if a given number is positive, zero, or negative.

(4). Construct a flowchart for the process of finding the depth to which a spherical metal ball is submerged under water, given the density of the metal, the outer radius, and the wall-thickness of the ball. Assume the density of water is 1000 kg/m^3.

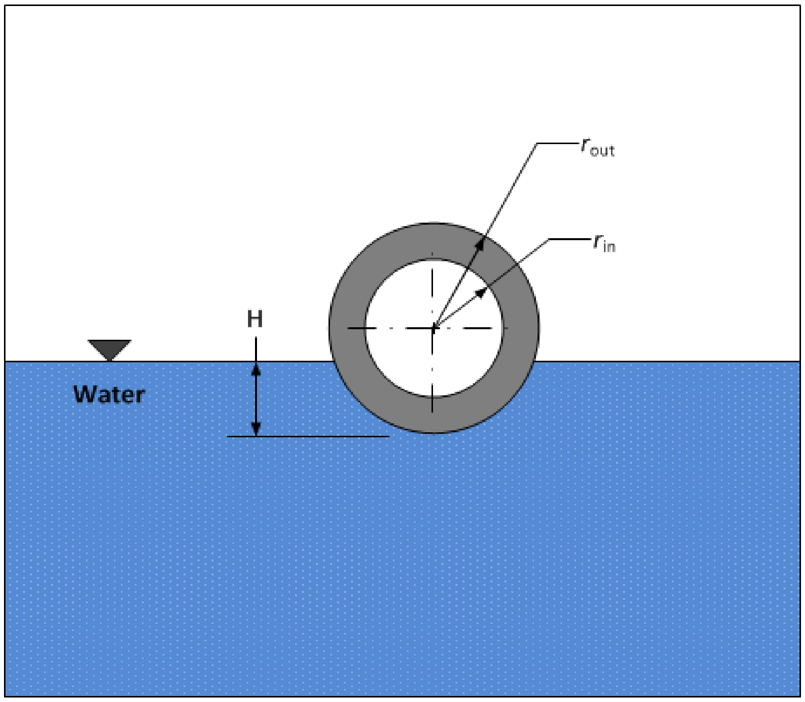
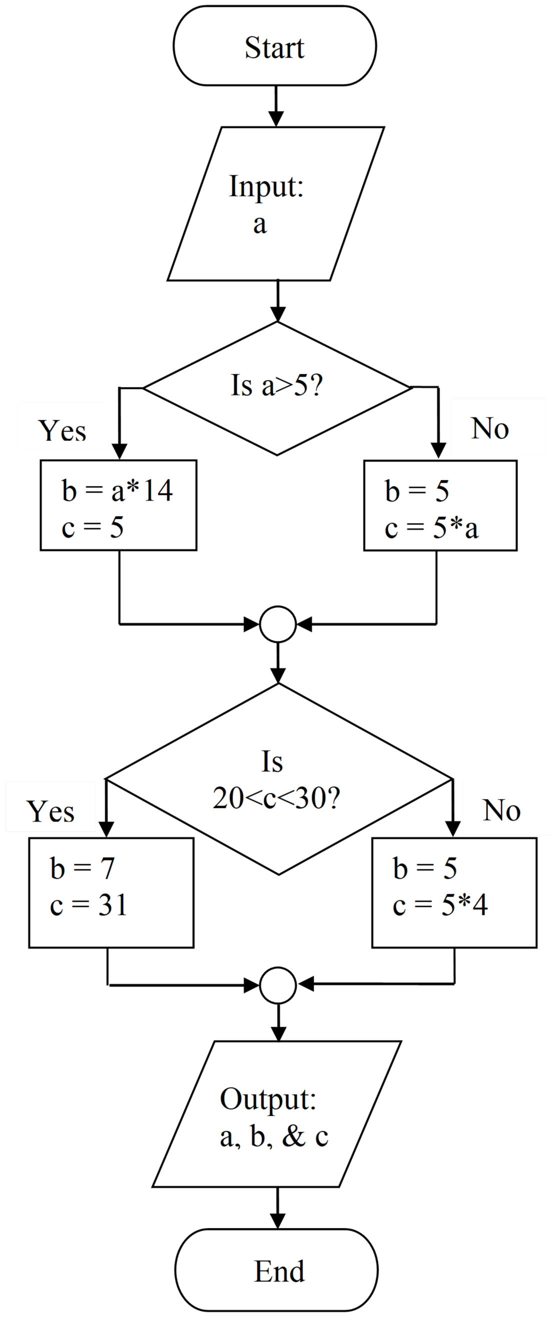


Figure A – Metal ball floating in water.

Hint: Apply Archimedes principle. The volume of the ball under water for a given depth, H is

$$V = \frac{\pi H^2 \left(3r_{\text{out}} - H\right)}{3}$$

(5). Convert the following flowchart into a MATLAB program.


Start
Input:
a
Is a>5?
Yes
No
b = a*14
c = 5
b = 5
c = 5*a
Is
20<c<30?
Yes
No
b = 7
c = 31
b = 5
c = 5*4
Output:
a, b, & c
End

Module 6: PROGRAM DESIGN AND COMMUNICATION

Lesson 6.2 – Pseudocode

Learning Objectives

After reading this lesson, you should be able to:

- *write your own pseudocode,*
- *write a MATLAB program based on reading a pseudocode.*

What is a pseudocode?

A pseudocode is an informal outline of a program or computer process that is written using a mixture of computer language syntax and English conventions. When a programmer writes a pseudocode, it is intended for the use of other programmers or users, and not for a computer to read or process. Similarly, pseudocodes are used in the realm of numerical methods to communicate ideas, without having to write out long numerical equations. There are no rules for what you can or cannot state in a pseudocode, but the code needs to be easily readable, therefore subroutines and formal variable names are often omitted.

To keep the pseudocode organized, a commonly accessible word processor should be used. This makes editing, opening, and sending the pseudocode easy. Remember, the pseudocode should be easy to read and user-friendly.

When planning out code, you can also write some quick and less formal pseudocode. In this case, handwritten pseudocode is perfectly fine. As you become a more experienced programmer, these tools will become more useful and intuitive.

How are pseudocodes used?

A good pseudocode is used to outline a specific order of events to solve a problem, and hence, bridging the gap between the problem and the solution. To solve any problem, one must first clearly identify the problem, next determine the methods to solve the problem, and finally, solve the problem. Code can generally be written faster and communicated better with the use of pseudocode(s) to design the structure of the program and outline the operations to be conducted. A pseudocode should usually be written without bias to any programming language. Keep in mind that programming languages do not always share the same syntax for the same task. For this reason, when developing a pseudocode, try to state the process desired and not the syntax needed to complete the process. Leave it to the reader to decide what syntax to use to complete the process. Example 1 shows how a pseudocode is developed.

Example 1

Your boss has given you the MATLAB program shown in Figure 1. He tells you that the program will be converted to Python by another worker, and needs you to develop a pseudocode for the Python worker to follow. Be sure to avoid using MATLAB specific syntax in the pseudocode, as MATLAB and Python have different syntax.

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To find the value of a derivative at a point
fprintf('To find the value of a derivative at a point.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
%Creating a symbolic variable, x. Must be defined here to avoid undefined
%    variable error from MATLAB.
syms x
y = input('Function of x to have first derivative taken: ');
xVal = input('Where do you want to find the slope? ');

%------------------------------ SOLUTION ----------------------------
dydx    = diff(y,x,1);        %Finding the first derivative
dydxVal = subs(dydx,x,xVal);  %Evaluating y'(x) at xVal

%------------------------------ OUTPUTS ------------------------------
fprintf('\nOUTPUTS\n')
fprintf('The value is %g\n',dydxVal)
```

Solution

The pseudocode:

Find the value of the derivative of a function at a point.

1. Clear all windows and workspace variables.
2. Display the purpose of the program.
3. Define x as a symbolic variable.
4. Prompt user to enter the desired function.

Enter as a function of x.

5. Take the first derivative of input function with respect to x.
6. Prompt user to enter the point where the derivative needs to be evaluated.
7. Substitute/replace x with the number.

Substitute the value for x where derivative needs to be found.

8. Display value of the derivative of the function at a given value of x.
9. End program.

Note that the pseudocode is an informal outline containing a mixture of English and programming syntax. This makes the process easier for any reader to follow and to interpret. Even if the reader does not understand MATLAB, they should still be able to understand the process of the program. The pseudocode in Example 1 is fairly detailed and could be made more compact. However, be aware, that a fine line separates an over-detailed pseudocode from an under-detailed one, and that it is the responsibility of the writer to provide enough details to outline the process clearly.

How can I convert a pseudocode for a problem into a program?

If you are given a problem, you can write a pseudocode by following the logic of how the problem would be solved. The pseudocode is now a roadmap to writing the MATLAB program. Fully read the pseudocode to understand the objective of the program. Once you have read the pseudocode, re-read it, this time marking the inputs and corresponding outputs. Also, you should be taking a mental (or written) inventory of the commands and functions needed to accomplish the objective. Look at Example 2 to see how a pseudocode is written for a simple problem and then turned into a program.

Example 2

You are given the task of writing a MATLAB program for finding the value of the definite integral based on a customer's specifications.

$$\int_a^b f(x)\,dx$$

The specifications state that the user is to be prompted to enter all program inputs (they are not interested in seeing the m-file) and that only the value of the integral should be displayed.

a. Write a pseudocode that outlines the problem.

b. Solve the problem using MATLAB.

Solution

a. The pseudocode reads:

Find the area under a curve.

- Clear all windows and workspace variables
- Display the purpose of the program
- Input (as prompted)
  - Function, $f(x)$
  - The lower limit of integration, a
  - The upper limit of integration, b
- Integrate ($f(x)$, x, a, b)
- Output
  - The area under a curve

b. With the pseudocode provided in part (a), the following m-file has been developed. Note the order and process of the program, and compare the m-file to the pseudocode.

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find the area under a curve
fprintf('To find the area under a curve.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
%Creating a symbolic variable, x. Must be defined here to avoid undefined
%    variable error from MATLAB.
syms x
func  = input('Enter the function to be evaluated: ');
lower = input('Enter the lower bound: ');
upper = input('Enter the upper bound: ');

%----------------------------- SOLUTION ----------------------------
%Integrating the function with respect to x
intEval = int(func,x,lower,upper);
areaInt = double(intEval);

%----------------------------- OUTPUTS -----------------------------
fprintf('\nOUTPUTS\n')
fprintf('The area under the curve is %g.\n',areaInt)
```

Command Window Output Example 2

```
PURPOSE
To find the area under a curve.

INPUTS
Enter the function to be evaluated: x^3+2*x^2-7*x
Enter the lower bound: 4
Enter the upper bound: 12

OUTPUTS
The area under the curve is 5781.33.
```

Multiple Choice Quiz

(1). A pseudocode is

(a) a mixture of computer language and English.

(b) computer language in an outline form.

(c) English that is translated to a computer language

(d) a waste of time.

(2). Pseudocodes are written by

(a) humans for computers to read

(b) humans for humans to read.

(c) computers for computers to read.

(d) computers for humans to read.

(3). The format for a typical pseudocode is

(a) paragraph form.

(b) MATLAB comment form.

(c) a collection of abbreviations.

(d) outline form.

(4). Computer-language specific names should be

(a) used as little as possible in a pseudocode.

(b) okay to use in a pseudocode.

(c) entirely avoided in a pseudocode.

(d) included as much as possible in a pseudocode.

(5). When writing pseudocode, the format can include ________ comments per line.

(a) one

(b) two

(c) three

(d) as many as you want

Problem Set

(1). Write a pseudocode for a program that outputs the surface area and volume of a sphere where the input is the radius of the sphere.

(2). Write a pseudocode for a program that outputs a person's BMI where the inputs are the person's weight in lbs and height in inches. The BMI formula is,

$$\text{BMI} = \frac{weight\ (lbs)}{[height\ (in)]^2} \times 703$$

(3). Convert the MATLAB program shown below into a pseudocode.

```
clc
clear

%--------------------------- PURPOSE ---------------------------
fprintf('PURPOSE\n')
%To find the derivative of a function
fprintf('To find the derivative of a function.\n\n')

%--------------------------- INPUTS ----------------------------
fprintf('INPUTS\n')
syms x
y = input('Input the function to differentiate:  ');
xVal = input('What value do you want to substitute for x?  ');

%--------------------------- SOLUTION --------------------------
dydx = diff(y,x,1);
dydxVal = subs(dydx,x,xVal);

%--------------------------- OUTPUTS ---------------------------
fprintf('\nOUTPUTS\n')
fprintf('The derivative of the function is %s.\n',char(dydx))
fprintf('The value of the y''(%g) is %g.\n',xVal,double(dydxVal))
```

Module 6: PROGRAM DESIGN AND COMMUNICATION

Lesson 6.3 – Writing Better Code

Learning Objectives

After reading this lesson, you should be able to:

- *use techniques for evaluating and decreasing computational time,*
- *provide useful information in comments to other programmers,*
- *organize your code using proper indenting,*
- *choose inputs and outputs wisely,*
- *create your own error and warning messages to users.*

This lesson is about how to write code that is efficient in the computational power it demands, the time it takes you to write (and especially edit it), and the time it takes for someone else to understand/edit your code.

Just like traditional writing where we use punctuation, paragraphs, lists, fonts, etc. to improve the "visual performance" of our work, the same necessity applies to code. Syntax is the punctuation and grammar. Using appropriate spacing and comments helps the reader categorize and fully understand the code. In MATLAB, using sections helps someone navigate and debug the code.

How can I improve my code for computational efficiency?

Use more efficient functions - Not all functions are created equal. Remember, they are algorithms that perform some function: some set of steps. In some cases, changing the function (algorithm) you use in your code will significantly affect the time it takes to run your program. For example, `line()` is generally computationally less demanding than `plot()`. Why not always use the more efficient function?

Write more efficient code - One example of making your code more efficient is to try to reduce the number of loop iterations in it (if applicable). For instance, you might change the structure of your code to get rid of a nested loop or use a binary search. We will not cover this in-depth here, but know that it is important in contexts where computational efficiency is crucial. The salience of computational efficiency is relative, which means you might have a "big" program/large dataset *or* a "small" computer.

MATLAB Profiler Tool - To help identify specific areas for scrutiny in your program, you can use the MATLAB Profiler tool (see more on their Profile to Improve Performance page). This tool can be run by clicking the "Run and Time" button in the Editor tab next to "Run" (see Figure 1).

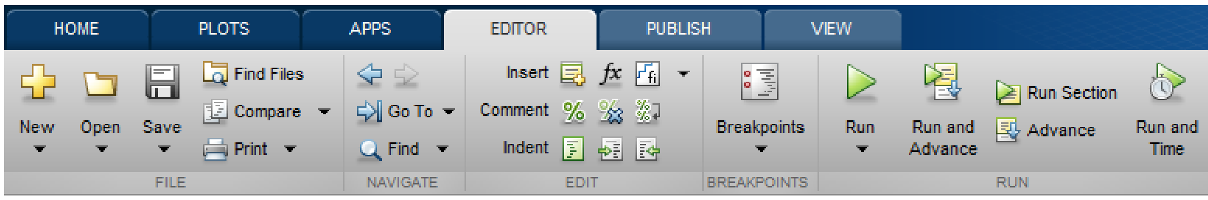
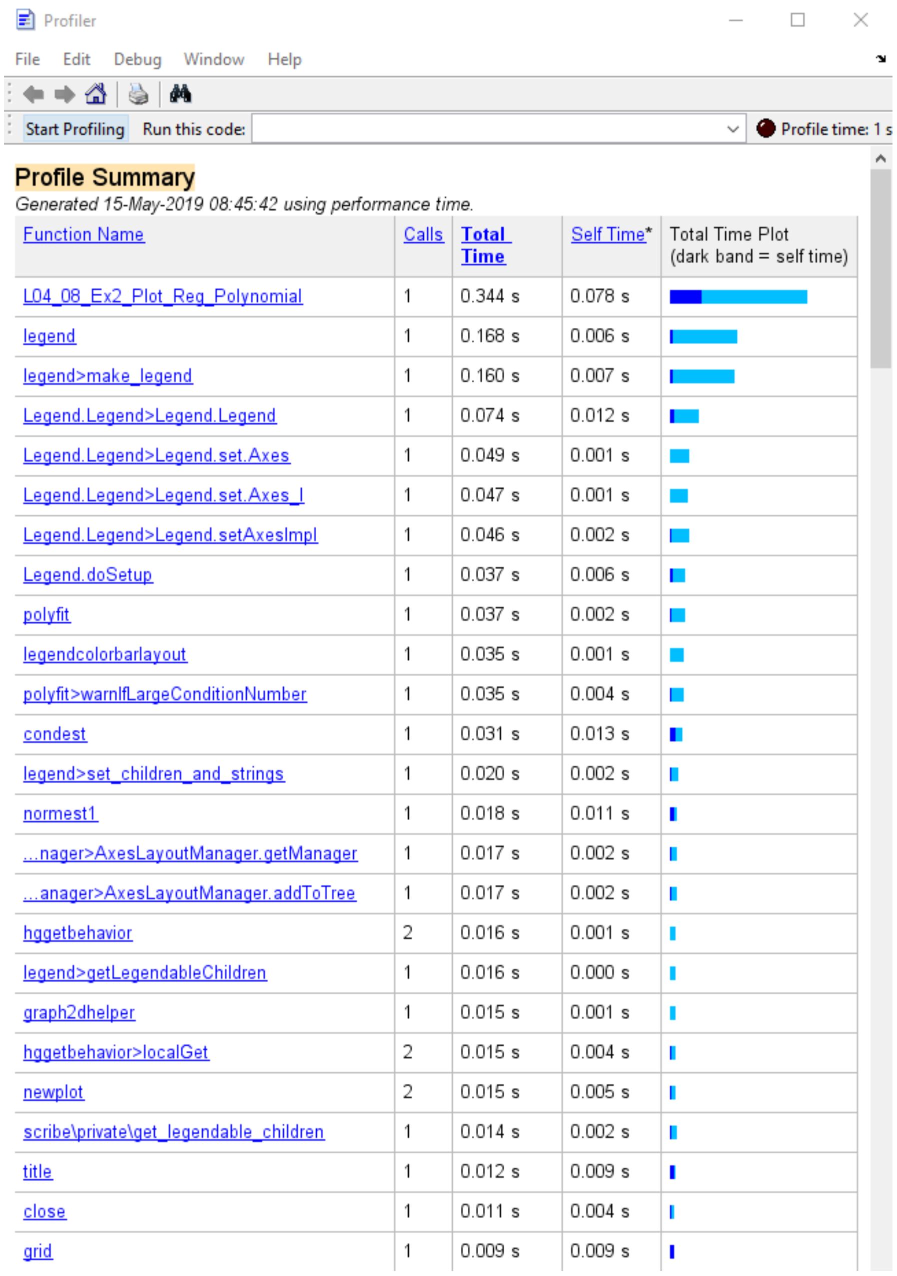


Figure 1: The Editor tab highlighting the "Run and Time" button for the MATLAB Profiler.



Profile Summary

Generated 15-May-2019 08:45:42 using performance time.

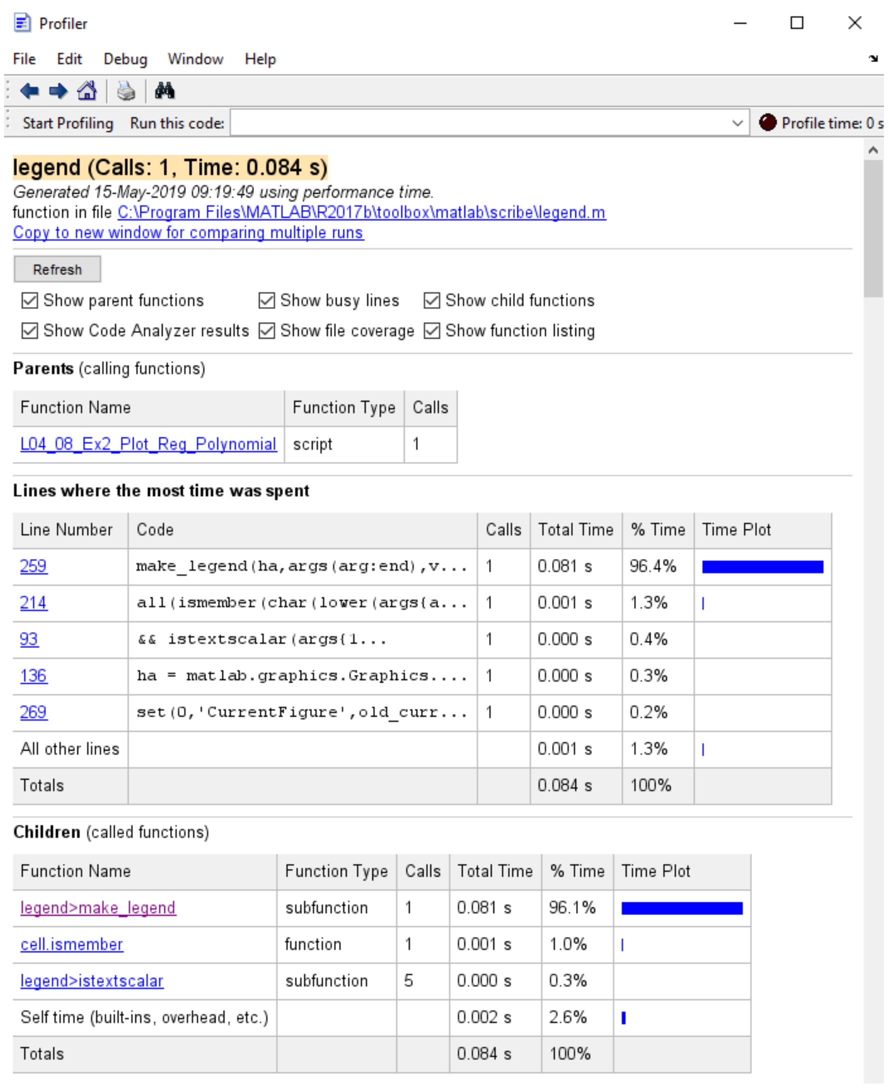
Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
L04_08_Ex2_Plot_Reg_Polynomial	1	0.344 s	0.078 s	
legend	1	0.168 s	0.006 s	
legend>make_legend	1	0.160 s	0.007 s	
Legend.Legend>Legend.Legend	1	0.074 s	0.012 s	
Legend.Legend>Legend.set.Axes	1	0.049 s	0.001 s	
Legend.Legend>Legend.set.Axes_I	1	0.047 s	0.001 s	
Legend.Legend>Legend.setAxesImpl	1	0.046 s	0.002 s	
Legend.doSetup	1	0.037 s	0.006 s	
polyfit	1	0.037 s	0.002 s	
legendcolorbarlayout	1	0.035 s	0.001 s	
polyfit>warnIfLargeConditionNumber	1	0.035 s	0.004 s	
condest	1	0.031 s	0.013 s	
legend>set_children_and_strings	1	0.020 s	0.002 s	
normest1	1	0.018 s	0.011 s	
...nager>AxesLayoutManager.getManager	1	0.017 s	0.002 s	
...anager>AxesLayoutManager.addToTree	1	0.017 s	0.002 s	
hggetbehavior	2	0.016 s	0.001 s	
legend>getLegendableChildren	1	0.016 s	0.000 s	
graph2dhelper	1	0.015 s	0.001 s	
hggetbehavior>localGet	2	0.015 s	0.004 s	
newplot	2	0.015 s	0.005 s	
scribe\private\get_legendable_children	1	0.014 s	0.002 s	
title	1	0.012 s	0.009 s	
close	1	0.011 s	0.004 s	
grid	1	0.009 s	0.009 s	

Figure 2: Output window of the MATLAB Profiler run on Example 2 from Lesson 4.8.

In Figure 2, we see an example output of the MATLAB Profiler. The time to run the whole m-file is given on the first line: 0.344 seconds in this case. We can recognize some of the entries like `title()`, `legend()`, `close`, and `polyfit()`, while others are not familiar. The unfamiliar lines are functions/commands that execute as a direct result of the code we wrote in our m-file.

Looking at the second line of the Profiler output, we see that the `legend()` is taking the longest to complete in this program (although still very short). Clicking on the function name in the Profiler will open the Profile Detail Report window (see Figure 3). This will give specific details (down to the line of code) on run time for that function.

You can see in Figure 3 that a summary of the lines that took the longest to run is given as well as the sub-functions or "child" functions. We will end our discussion of the MATLAB Profiler and computational efficiency for now as more programming knowledge and experience are needed to dig deeper into the topic. The principal thing to remember is that there are easy-to-use tools in MATLAB that can help you make your code run faster.



legend (Calls: 1, Time: 0.084 s)
Generated 15-May-2019 09:19:49 using performance time.
function in file C:\Program Files\MATLAB\R2017b\toolbox\matlab\scribe\legend.m
Copy to new window for comparing multiple runs

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
L04_08_Ex2_Plot_Reg_Polynomial	script	1

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
259	`make_legend(ha,args(arg:end),v...`	1	0.081 s	96.4%	▬
214	`all(ismember(char(lower(args(a...`	1	0.001 s	1.3%	|
93	`&& istextscalar(args{1...`	1	0.000 s	0.4%	
136	`ha = matlab.graphics.Graphics....`	1	0.000 s	0.3%	
269	`set(0,'CurrentFigure',old_curr...`	1	0.000 s	0.2%	
All other lines			0.001 s	1.3%	|
Totals			0.084 s	100%	

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
legend>make_legend	subfunction	1	0.081 s	96.1%	▬
cell.ismember	function	1	0.001 s	1.0%	|
legend>istextscalar	subfunction	5	0.000 s	0.3%	
Self time (built-ins, overhead, etc.)			0.002 s	2.6%	▮
Totals			0.084 s	100%	

Figure 3: The Profile Detail Report for the `legend()` function, which was called in an example program.

How does hardcoding impact a program?

If you are new to programming, you have probably never heard of “hardcoding”. It occurs when unnecessary changes inside the body of the code *are* necessary when the inputs of the program are changed. If a code is written well, only the inputs need to be changed for the program to still be valid (whether that relates to math, syntax, or something else). It is especially problematic if someone else is trying to use your code, but does not know that it only works correctly for specific inputs. For example, you have inputs for the dimensions of a beam but not for the material properties of the beam like Young's modulus, which you hardcoded later in the program.

What are some tips for good comments and spacing?

It is always important to make proper use of comments in your programs. A few reasons to use good comments are that they:

- help you understand and write better code,
- help you debug the code as you are writing the program,
  - Track issues in the code. It can help to use a unique, searchable keyword like "FIX".
  - Remember why you did "X" a week ago.
  - Provides organization, so you can more easily isolate problems.
- document the code for future reference (for yourself *and* others).

Now, a few ways to write good comments are:

- Organize your code with "stand-out" headings (for longer pieces of code ~100+ lines),
  - Do not overuse these headings as that would defeat the purpose,
  - Put a short, simple description,
- Leave detailed notes on things that are especially complicated,
- Leave a comment explaining the purpose of each variable when you define it for the first time.

MATLAB Code example1.m

```
%Below are some examples of how to format headings that stand out from your code and
%    help organize the different parts.

%%%%%%%%%%%%%%%%%%  DESCRIPTION  %%%%%%%%%%%%%%%%%%



%===================================================
%                   DESCRIPTION
%===================================================



%----------------- DESCRIPTION -----------------
```

Why does proper code indenting matter?

You may have noticed that certain blocks of code are indented. There are a number of different cases where this is standard practice, but the two most common are for conditional statements, which you have seen in Module 5, and for loops, which you will see in Module 8. Although MATLAB does not require indenting, some programming languages (like Python) do require proper indenting. In any case, using proper indenting is standard practice in MATLAB because it makes your program more readable and clearer.

Fortunately, MATLAB makes indenting easy with “smart indenting” tools. Just highlight the portion of code you want it to fix and click the appropriate button in the toolbar (see Figure 1).

What are some tips for choosing inputs and outputs?

Choosing appropriate inputs and outputs to your program (and, later in the course, your functions) is a balance between giving the user enough control to make the program useful, but not so many as to overwhelm them. For example, if you are writing a program to calculate the weight of an object (w = mg), obviously weight would be your output. You would want to set the mass of the object as an input but probably not the acceleration due to gravity. However, if you worked at NASA, both the mass and the acceleration due to gravity would be good inputs.

On the other side of things, you do not want to output too much information. For instance, you should avoid outputting too many intermediate steps of the program (as a finished product). If you are writing a program that checks to see whether your item will fit in a certain shipping box, you do not need to report the checks for each of the three dimensions. Just report "fits" or "item too large".

The key to making good choices is to think about the uses for your code and the audience (users). There are no general rules other than this on making a decision on what you will require from and report to the user.

What are some tips for thinking ahead in when designing my program?

So far in this lesson, we have talked about thinking ahead when writing code in several contexts such as writing informative comments and choosing the appropriate inputs and outputs. However, this is not the extent of planning that should go into your code. As you mature as a programmer, you will start to pick up more things that you will consider as you write a new program. Some common examples of this have to do with numbering conventions. The "Y2K" crisis was caused because no one implemented a simple and "obvious" case for a change in millennia. The problem arose from only a year with its last two digits (e.g., 97, 98, 99, 00).

Not only should we try to anticipate a user's needs, but we should write programs as robustly as possible whenever possible. That means expecting the unexpected. For example, use the `else` conditional clause with care as "else" encompasses a lot of things (i.e., *everything* else). Check user inputs for validity (correct data type, numeric range/sign, etc.).

Multiple Choice Quiz

The content of this page is intentionally blank

Problem Set

The content of this page is intentionally blank

Module 7: FUNCTIONS

Lesson 7.1 – User-Defined Functions

Learning Objectives

After reading this lesson, you should be able to:

- write your own functions in MATLAB,
- use the rules for creating and naming functions,
- write functions in a program m-file in MATLAB,
- write multiple functions in one function m-file.

What is a function?

You already know what a function is because we have been using MATLAB built-in functions through this whole book! Simply put, a function is a piece of code with explicit input and output variables. A function performs a task for you. For example, the function `sin()` takes a number in radians as its input and returns the sine of the input. It is executed whenever/wherever it is called in an m-file. Of course, the sine of a number is not magically found: there is some code behind the scenes. In this module, we will show you how to create and use your own custom functions.

The functions you make are called "user-defined" functions. These are no different conceptually than the "built-in" MATLAB functions you have been using so far in the course. We use "user-defined functions" as a shorthand to indicate it is a custom function that we write ourselves.

The lines of code that are below the function statement (see Example 1b) are called the body of the function, and contain standard MATLAB code. This is where all processes, operations, or logic tests are placed in the function.

What are the naming rules for functions in MATLAB?

Similar to naming m-files and variables, there are rules for what you can name functions. In fact, they are the same rules as those for m-files and variables, which you can review in Lesson 2.1. In addition to those rules, the name of the function must be the same when it is defined and when it is called. For example, if you defined a function and name it `myFunc()`, then you must use the same name when you call it (`myFunc()`).

How can I create functions in MATLAB?

User-defined functions exist in many popular programming languages. MATLAB functions, just like in other languages, have input and output variables, which need to be defined both when the function is created and when it is called.

Some general steps to follow when you are creating functions in MATLAB are given here. These steps apply to functions that are saved in a separate m-file as in Examples 1, 2, and 4.

1. Choose input and output variables for your function. Think about how your function will be used.
2. Save the function in the same folder as your main m-file.
3. Write your code in the function.
4. Make sure all changes in your function m-file are saved so they will be reflected when you call the function.
5. Define the input and output variables in your main m-file.
6. Call the function in the main m-file, and test run the main m-file.

When beginning to write a function, one may not know exactly how many outputs or inputs the function will have. To help determine this, it is a good idea to develop a flowchart (Lesson 6.1) or pseudocode (Lesson 6.2) of the process before making the function.

The first method that we cover for creating a MATLAB function is to make two separate m-files: one for defining the function (Example 1b) and one for calling it (Example 1a). We need the m-file given in Example 1a because a function must be called in order to run (execute its code). You *cannot* run the function file simply by pressing "Run" from inside the function m-file like we have been doing for other m-files.

In this method, the function m-file must have the same name as the function. For example, the code in Example 1b must be saved in an m-file named `myTriangle.m`. Failing to follow this rule will return an error.

We present the m-file where the function is called (e.g., Example 1a) first because this is the way we are used to interacting with functions thus far. Next, the function definition m-file is shown (e.g., Example 1b).

Example 1a

Write and test a MATLAB function called `myTriangle()`. Given the length of the three sides of the triangle, it should find the area and perimeter of a triangle. If a, b, and c are the lengths of the three sides of the triangle, then the perimeter of the triangle is

$$P = a + b + c$$

and the area of the triangle is

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where s is the semi-perimeter of the triangle, and

$s = \frac{P}{2}$.

Test the program using inputs of $a = 3,\ b = 4,\ c = 5$.

MATLAB Code example1a.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find the perimeter and area of a triangle
fprintf('To find the perimeter and area of a triangle.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
a = 3;
b = 4;
c = 5;
fprintf('The lengths of the sides of the triangle are %g, %g, and
%g.\n\n',a,b,c)

%----------------------------- SOLUTION ----------------------------
%Calling a custom function we made (see Example 1b below)
[A,P] = myTriangle(a,b,c);

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The area of the triangle is %g.\n',A)
fprintf('The perimeter of the triangle is %g.\n',P)
```

Example 1b

This function is called in the program written in Example 1a. The problem statement is given there.

To write the function, let us first define the desired input and output variables.

Input variables:

- a = length a of triangle
- b = length b of triangle
- c = length c of triangle

Output variables:

- `area` = area of triangle
- `perimeter` = perimeter of triangle

The function, `mytriangle()`, will hence be defined,

- `function [area,perimeter] = myTriangle(a,b,c)`

```
MATLAB Code                                                    example1b.m
function [area,perimeter] = myTriangle(a,b,c) %Defining function

%Coding the body of the function
perimeter = a+b+c;                 %Perimeter of the triangle
s = perimeter/2;                   %Semi-perimeter of the triangle
area = sqrt(s*(s-a)*(s-b)*(s-c));  %Area of the triangle
```

```
Command Window Output                                           Example 1b
PURPOSE
To find the perimeter and area of a triangle.

INPUTS
The lengths of the sides of the triangle are 3, 4, and 5.

OUTPUTS
The area of the triangle is 6.
The perimeter of the triangle is 12.
```

Note that all the statements have been suppressed from showing in the Command Window in the function `myTriangle()` (see Example 1b). Unsuppressed statements can confuse the user of a function, who may be using it transparently just as one uses the MATLAB functions such as `cos()`, `int()`, etc.

The result from the function `myTriangle()` is returned to the m-file where the function is called (see Example 1a), which can then be displayed to the Command Window if the user chooses to.

To avoid confusion, follow these five rules for developing a function:

1. always save your function m-file as the function name,
2. do not assign inputs in the function m-file,
3. never use `clc`, `clear`, or `close` commands in the function,
4. suppress all statements by placing the ; at the end of each statement,
5. avoid displaying information (using the `disp()` or `fprintf()`) inside the function m-file.

Although the variables within the function file must be consistent, they do not have to be the same when you call the function. Think about any MATLAB function you have called: `sin()`, for example. You have no idea what the variables names are inside the predefined function `sin()`, and you could use any input or output variable names when you called the function. That is, both `output1 = sin(input1)` or `sinVec = sin(vec)` or any other combination of valid variable names will work. However, input and output variable names must be consistent throughout the function m-file where it is defined (see Example 2b et al.).

MATLAB also needs to know where to find the function m-file when you call it. The easiest way to achieve this is to put both m-files in the same folder, and this is typically what you should do. If you need to access the function file often in different programs, look at the documentation for adding the function directory (folder) to a MATLAB path. This means the m-files will not have to be in the same folder, and MATLAB will automatically “find” them.

✓ ***Important Note:*** Function m-file must be in the same location as the main m-file, or it must be added to a MATLAB path.

Example 2a

Write a program that calls a function, which determines the sign of a number (positive, negative, or zero).

Test the program using an input of `num = 3`.

MATLAB Code example2a.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To determine the sign of a number
fprintf('To determine the sign of a number.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
num  = 3;  %Defining input to function
fprintf('The number to test is %g.\n\n',num)

%----------------------------- SOLUTION ----------------------------
%Calling a custom function we made (see Example 2b below)
sign = numberSign(num);

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The sign of the number is %s.\n',sign)
```

Example 2b

This function is called in the program written in Example 2a. The problem statement is given there.

MATLAB Code example2b.m

```
function sign = numberSign(num)

%Take the input "num" and conditionally check the sign
if num > 0
    %If the input number is greater than zero, return “positive”
    sign = 'positive';
elseif num < 0
    %If the input number is less than zero, return “negative”
    sign = 'negative';
else
    %If the input number is zero, return “number is 0”
    sign = 'number is 0';
end
```

Command Window Output Example 2b

```
PURPOSE
To determine the sign of a number.

INPUTS
The number to test is 3.

OUTPUTS
The sign of the number is positive.
```

Can I define functions in the program m-file?

For the MATLAB version R2016b and later, MATLAB allows us to define functions in the same m-file as the program we call them in. That is, we can define and call a function in the same m-file as seen in Example 3 (we do not have to create a separate file). This is the second method for defining MATLAB functions that we will cover in this lesson. An `end` is required for functions that are defined in a program m-file (see Example 3). It can also be required in other cases (see Example 4b). To be safe, you can default to always putting an `end` to terminate your functions.

✓ ***Important Note:*** Function definitions must be at the end of the m-file for functions defined in a program file (functions defined and called in the same m-file).

Example 3

In 1998, the federal government developed the body mass index (BMI) to determine healthy weights. Body mass index is calculated as 703 times the weight in pounds divided by the square of the height in inches. The obtained number is then **rounded off** to the nearest whole number (***Hint:*** 23.5 will be rounded to 24; 23.1 will be rounded to 23; 23.52 will be rounded to 24). The criterion for the weight category is simplified for the example as is given below.

$BMI < 19$ – Underweight

$19 \leq BMI \leq 25$ – Healthy Weight

$BMI > 25$ – Overweight

Write a function that accepts weight (lbs) and height (in) as input variables, and outputs the BMI as an integer, weight category as a string (underweight, healthy weight or overweight), and the target weight (lbs) as an integer. The target weight of an underweight person would correspond to a BMI of 19, and that of an overweight person to a BMI of 25. No `fprintf()` or `disp()` statements should be in the function.

```
function [BMI,category,target_weight] = myBMI(w,h)
```

Test the program for an individual who weighs 185 lbs and is 75 inches tall.

MATLAB Code example3.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To calculate BMI and advise on a target weight
fprintf('To calculate BMI and advise on a target weight.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
weight = 180;  %lbs
height = 75;   %in
fprintf('The input weight is %g.\n',weight)
fprintf('The input height is %g.\n\n',height)

%------------------------------ SOLUTION -----------------------------
%Call the function and store the outputs in variables
[BMI,category,targetWeight] = myBMI(weight,height);

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')

fprintf('The calculated BMI is %g, which is considered %s.\n',BMI, category)
fprintf('Based on this, the target weight is %g.\n',targetWeight)
```

MATLAB Code (continued) example3.m

```
%----------------------- FUNCTION DEFINITION -----------------------
%Defining our function in the program m-file (compatible with R2016b and
%    newer). Note how we define and call our function in the same m-file.
function [BMI,category,targetWeight] = myBMI(w,h)
BMI = round(w/h^2*703);
if BMI<19
    category = 'underweight';
    targetWeight = round(19*h^2/703);
end
if BMI>=19 && BMI<=25
    category = 'healthy';
    targetWeight = round(w);
end
if BMI>25
    category = 'overweight';
    targetWeight = round(25*h^2/703);
end

%Note, this closing “end” is required for functions defined in a program
%    m-file.
end
```

Command Window Output Example 3

```
PURPOSE
To calculate BMI and advise on a target weight.

INPUTS
The input weight is 180.
The input height is 75.

OUTPUTS
The calculated BMI is 22, which is considered healthy.
Based on this, the target weight is 180.
```

The choice of whether to put the function in a program m-file (define and call in the same m-file) or define the function in its own separate m-file is completely up to you (provided your version of MATLAB supports it). One of the reasons you might choose to define the function in a separate m-file is if your program is already long and/or the function will be used by other programs.

Another function organization option is given in Example 4b where we define multiple functions in one function m-file: again, only for convenience. Note that these functions, such as `triangleArea()`, are called within the function `areaFunction()` in the same m-file. Therefore, calling `areaFunction()` will call the appropriate area calculation function, such as `triangleArea()`, and execute its code.

Example 4a

Write a program that calls a function, which finds the area of a given shape. The program should work for three different shapes: triangle, square, and circle. Inputs to the program and the function are a string containing the name of the shape and its dimensions.

Test the program for a triangle with sides of lengths 3, 4, and 5.

MATLAB Code example4a.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find the area of a shape
fprintf('To find the area of a shape.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
shapeInput = 'triangle';  %The shape we want to find the area of
dimensions = [3 4 5];     %The dimensions of the shape
fprintf('The %s has lengths of sides of %g, %g, and %g\n\n', ...
         shapeInput,dimensions(1),dimensions(2),dimensions(3))

%----------------------------- SOLUTION ----------------------------
%Call the function and store output
areaOut = areaFunction(shapeInput,dimensions);

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The area of the %s is %s.\n',shapeInput,num2str(areaOut))
```

Example 4b

This function is called in the program written in Example 4a. The problem statement is given there. This m-file is still saved in the usual way as `areaFunction.m`.

Note that all the functions shown below are in the same m-file. They are split here only for formatting.

MATLAB Code example4b.m

```
function area = areaFunction(shape,dim)

if strcmp(shape,'triangle') %strcmp() compares two strings and returns
                            %    boolean (true/false) value
    area = triangleArea(dim);
elseif strcmp(shape,'square')
    area = squareArea(dim);
elseif strcmp(shape,'circle')
    area = circleArea(dim);
else
    area = 'Unknown';
    disp('Your shape is not supported.')
end
end
%Note that "end" is required here to delimit the end of the body of each
%    function.


function areaT = triangleArea(dimNested)
%Perimeter of the triangle
perimeter = dimNested(1)+dimNested(2)+dimNested(3);
%Semi-perimeter of the triangle
s = perimeter/2;
%Area of a triangle is given by A = sqrt(s*(s-a)*(s-b)*(s-c))
areaT = sqrt(s*(s-dimNested(1))*(s-dimNested(2))*(s-dimNested(3)));
end


function areaS = squareArea(side)
%Area is given by A = a^2
areaS = side^s;
end


function areaC = circleArea(r)
%Area is given by A = pi*r^2
areaC = pi*r^2;
end
```

Command Window Output Example 4b

```
PURPOSE
To find the area of a shape.

INPUTS
The triangle has lengths of sides of 3, 4, and 5

OUTPUTS
The area of the triangle is 6.
```

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Create a MATLAB function	`function`	`function out = name(in);\ out = 4*in; end`

Multiple Choice Quiz

(1). To write a function named `myOwnSin()`, the function m-file should be named as

(a) anything I want

(b) `myOwnSin.m`

(c) `myOwnSinfunction.m`

(d) `sin.m`

(2). The following function is written and saved as an m-file called `multmine.m`

```
function c = multmine(a,b)
c = a*b;
```

The function is then called in a separate m-file as follows

```
clc
clear
a = 3;
t = 7;
multmine(a,t)
```

The Command Window output is

(a) `Undefined function or variable.`

(b) `3`

(c) `7`

(d) `21`

(3). What is the cause of the error that is returned when the following MATLAB code is run? The m-file is saved as `example.m`.

```
clc
clear
side = 6;
perimeter = perimeterSq(side)

function P = perimeterSq(len)
P = side*4;
```

(a) The required end is not at the end of the inline function `perimeterSq()`.

(b) The function is not called correctly.

(c) All of the inputs are not defined when the function is called.

(d) All of the above

(4). The following function is written and saved as an m-file called `multmine.m`

```
function c = multmine(a,b)
d = a*b;
```

The function is then called as follows

`multmine(3,7)`

The Command Window output is

(a) `Undefined function or variable.`

(b) `3`

(c) `7`

(d) `21`

(5). The following function is written and saved as an m-file called `myCheck.m`

```
function B = myCheck(a)
if a > 0
      B = 1;
end
if a < 0
      B = -1;
end
if a == 0
      B = 0;
end
```

The function is then called in a separate m-file as follows

```
clc
clear
T = 6;
soln = myCheck(T);
s = soln*2
```

The Command Window output is

(a) `Undefined function or variable.`

(b) `1`

(c) `2`

(d) `6`

Problem Set

(1). Write a MATLAB function that finds the area of a trapezoid given the length of the parallel sides and the perpendicular distance between the parallel sides. Name your function, `myTrap()`, where the function has three inputs, `side1`, `side2`, and `height`, and the output is `area`. Be sure not to use `fprintf()` or `disp()` inside the function m-file, and suppress all statements within the function. Test your function with at least two different sets of input values.

(2). A simply supported beam is loaded as shown in Figure A. Under the applied load, the beam will deflect vertically. This vertical deflection of the beam, V, will vary along the length of the beam from $x = 0$ to L, and is given by

$$V = \begin{cases} \dfrac{Pb}{6EIL}\left[(-L^2 + b^2)x + x^3\right], 0 < x < a \\ \dfrac{Pb}{6EIL}\left[(-L^2 + b^2)x + x^3 - \dfrac{L}{b}(x-a)^3\right], a < x < L \end{cases}$$

where,
x is the distance from the left end,
P is the load,
L is the length of the beam,
a is the location where the load P is applied,
E is the Young's modulus of the beam material, and
I is the second moment of area.

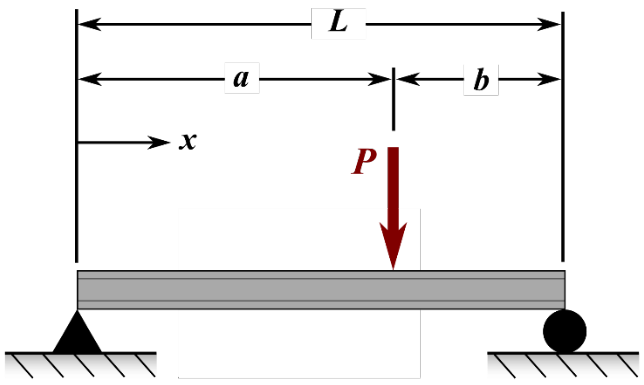


Figure A: Simply supported beam shown with applied load, P.

Complete parts (a) and (b).

a. Make a flowchart for the problem.

b. Write a MATLAB function that outputs the vertical deflection of the beam at a point of interest.

The function inputs are

1. distance from the left end to the point of interest, `x`,
2. length of the beam, `L`,
3. load, `P`,
4. location where the load P is applied, `a`,
5. Young's modulus of the beam material, `E`, and
6. second moment of area, `I`,

and output is,

1. the calculated deflection, `V`.

In your test m-file, display all of the inputs and outputs by using `fprintf()`, complete with explanation and reasonable format.

Run your test m-file for the following two input sets.

1. $x = 2.5,\ L = 5,\ a = 3,\ E = 30E6,\ I = 0.0256,\ P = 30$

2. $x = 4.05,\ L = 5,\ a = 3,\ E = 30E6,\ I = 0.0256,\ P = 30$

(3). The monthly payment on a car loan is given by the formula

$$PMT = \frac{LA * IPM}{1 - (1 + IPM)^{-NM}}$$

where,
PMT = monthly payment in dollars,
LA = loan amount in dollars,
IPM = interest rate in fraction per month (Note the units),
NM = number of monthly payments (Note the units).

Write a MATLAB function `myCar()` that outputs the monthly payment for buying a car based on the loan amount (dollars), length of loan (years) and interest rate (annual percentage rate). The three inputs to the function are

1. loan amount entered in dollars, `LA`,
2. length of loan entered in integer years, `NY`, and
3. interest rate entered in annual percentage rate (APR), `APR`.

The output is,

1. the monthly payment on the car.

Test your function at least three times for three different sets of inputs. Your test m-file should show the following in the Command Window.

- Loan amount in dollars,
- Length of loan in integer years,
- The interest rate in annual percentage rate,
- Monthly payment in dollars.

(4). The three principal stresses $\sigma_1, \sigma_2, \sigma_3$for the stress state

$$\left[\sigma_x,\ \sigma_y,\ \sigma_z,\ \tau_{xy},\ \tau_{yz},\ \tau_{zx}\right]$$

at a point are given by the solution of the nonlinear equation

$$\sigma^3 - J_3\sigma^2 - J_2\sigma - J_1 = 0$$

where,
$J_1 = \sigma_x\sigma_y\sigma_z + 2\tau_{xy}\tau_{yz}\tau_{zx} - \sigma_x\tau_{yz}^2 - \sigma_y\tau_{zx}^2 - \sigma_z\tau_{xy}^2$
$J_2 = \tau_{xy}^2 + \tau_{yz}^2 + \tau_{zx}^2 - \sigma_x\sigma_y - \sigma_y\sigma_z - \sigma_z\sigma_x$
$J_3 = \sigma_x + \sigma_y + \sigma_z$

Write a MATLAB function to find the principal stresses, if the stress state is given as a vector of elements. The output principal stresses should be a vector of $[\sigma_1,\ \sigma_2,\ \sigma_3]$.

(5). Using your knowledge of conditional statements, write a MATLAB function that evaluates the "hotness" level of water leaving a faucet. Prior to exiting the faucet, cold and hot water are mixed together to adjust the output water temperature, where the hot water level may be adjusted from 0 to 100%. The outlet temperature ($T_o(^oF)$) is rounded to the nearest whole number and may be found by

$$T_o = (H_w)\,165 + (1 - H_w)\,62$$

where H_w is the percentage of hot water used and is entered as a decimal (i.e., 95% is entered as 0.95). There are four "hotness" levels, which are:
- extremely hot water, $T_o \geq 142^oF$
- hot water, $110 \leq T_o < 142^oF$
- warm water, $85 \leq T_o < 110^oF\backslash$ - room temperature water, $T_o < 85^oF$
Save your function m-file as `outletWater.m`.

Your function has one input, which is

1. the percentage of hot water used (entered as a decimal),

and will have two outputs, which are

1. a string of characters describing the current water condition, and
2. the temperature of the water.

Test your function using a hot water input of 84% (entered as 0.84).

(6). United States citizens pay federal income tax, social security tax, and Medicare tax on their wages.

Federal Income Tax: According to the Internal Revenue Service, a single-status U.S. citizen will pay 2018 federal income taxes according to the following chart*.

10% on income between \$0 and \$9,525
12% on the income between \$9,526 and \$38,700; plus \$952.50
22% on the income between \$38,701 and \$82,500; plus \$4,453.50
24% on the income between \$82,501 and \$157,500; plus \$14,089.50
32% on the income between \$157,501 and \$200,000; plus \$32,089.50
35% on the income between \$200,001 and \$500,000; plus \$45,689.50
37% on the income over \$500,001; plus \$150,689.50

Social Security tax: In 2018, this is 6.2% on earnings up to \$128,400**
Medicare tax: In 2018, this is 1.45% on all earnings.** An additional 0.9% applies to individuals who earn over \$200,000.***

Complete the following:

(a). Construct a pseudo code for a MATLAB function to find a person's total tax dollars owed based on their income. Assume all income is taxable.

(b). Assume all income is taxable and write a function `myTaxes()` has an input of the person's income, income, and then outputs the

1. federal tax, `fedTax`,
2. social security tax, `ssTax`,
3. medicare tax, `medTax`, and
4. total taxes owed, `totalTax`.

In your test m-file, display all the inputs and outputs by using `fprintf()`, complete with explanation and reasonable format.

Test your function for the following incomes: \$32,000, \$85,000 and \$269,000.

(7). Using your knowledge of conditional statements, write a MATLAB function that determines the letter grade and overall percentage score for a student. The function must use the score breakdown as follows:

Table A: Student grade information.

Category	**Weight**	**Example Scores**
Homework	10%	88
Projects	12%	95
Tests	48%	76
Final exam	30%	91

The total score is rounded to the nearest whole number to calculate the letter grade as follows:

- $90 \leq A \leq 100$,
- $80 \leq B \leq 89$,
- $70 \leq C \leq 79$,
- $F \leq 69$

Save your function m-file as `letterGrade.m`.

There are three function inputs, which are the

1. homework score out of 800
2. test score out of 300
3. quiz score out of 80

There are two outputs which are

1. a string containing the student's letter grade, and
2. the overall percentage score.

Test your function with the following set of inputs:

Homework score of 620

Test score of 260

Quiz score of 62

References:

* Kelly Phillips Erb, "New: IRS Announces 2018 Tax Rates, Standard Deductions, Exemption Amounts And More"
<https://www.forbes.com/sites/kellyphillipserb/2018/03/07/new-irs-announces-2018-tax-rates-standard-deductions-exemption-amounts-and-more/#28e98d0d3133>

**U.S. Government, "Update 2019 – Social Security"
<https://www.ssa.gov/pubs/EN-05-10003.pdf>

*** U.S. Government, "Questions and Answers for the Additional Medicare Tax"
<https://www.irs.gov/businesses/small-businesses-self-employed/questions-and-answers-for-the-additional-medicare-tax>

Module 7: FUNCTIONS

Lesson 7.2 – Function Design and Communication

Learning Objectives

After reading this lesson, you should be able to:

- *add descriptions of your function for the help command,*
- *define your own custom errors and warnings in a function.*

How can I add a description for my function?

Remember the `help` command we learned about at the beginning of this course? This utility function also allows us to get a summary of information about any of our custom functions. MATLAB allows us to provide helper descriptions in the functions that we write so that we can reference back to them through the Command Window if needed. These function descriptions come in handy when trying to quickly remember what a function does and how to use it. The spacing and formatting of the description seen in Example 1 is optional but suggested for clarity. You can also include a brief function usage example within the function description!

The help text must come immediately after the function definition line. This is the only rule. However, it is standard to make the first word in the help description as the function name in all capital letters. This will make any other mentions of the function name bolded in the help text (see Command Window output for Example 1). In Example 1, the function is `myTriangle()`, so the first word of the second line is `"MYTRIANGLE"`.

Example 1

Add a description for the help command to `myTriangle()`. See Example 1 from Lesson 7.1 for the original problem statement and full solution.

MATLAB Code example1.m

```
function [area,perimeter] = myTriangle(a,b,c) %Defining function
% MYTRIANGLE This is a description of the function myTriangle(). You may
% want to include information about what the function does and its input
% and output variables.

perimeter = a+b+c;
s = perimeter/2;
area = sqrt(s*(s-a)*(s-b)*(s-c));
```

Command Window Input Example 1

```
>> help myTriangle

myTriangle This is a description of the function myTriangle(). You may
want to include information about what the function does and its input
and output variables.
```

How can I define errors and warnings inside my function?

We first learned about defining custom warning and error messages in Lesson 6.3: Writing Better Code (as well as many other helpful tips). Now, we will apply the `error()` and `warning()` functions inside a custom function that we write. The main difference for error handling in our custom functions is that you will need to foresee possible mistakes or misunderstandings that users might have. This can include mistakes a user might make regarding general information like mathematics or domain/problem-specific knowledge. For example, if you know that it is not possible to have a negative measure of length, you may want to check that the length input to your function is not a negative number and display a warning message if the length given is less than zero. As a side note, `error()` and `warning()` functions are not exclusive to our custom functions but can be implemented anywhere in your MATLAB code.

As seen in Example 2, if the user triggers a `warning()`, a custom message will be displayed, and the running program will continue to execute. However, if the user triggers an `error()`, the program will halt, and the error message will be displayed.

Example 2

Write a function that calculates the area of a square. Inside the function, check that the given dimension of the square is valid and return a warning if it is negative. Also check to see if the input shape is supported and return an error if it is not. In this case, only squares are supported.

Solution

MATLAB Code example2.m

```
clc
clear

%---------------------------- PURPOSE ----------------------------
fprintf('PURPOSE\n')
%To find the area of a square
fprintf('To find the area of a square.\n\n')

%---------------------------- INPUTS -----------------------------
fprintf('INPUTS\n')
shapeInput = 'square';  %The shape we want to find the area of
side = 6;               %The dimensions of the shape
fprintf('The %s has input dimension of %g.\n\n',shapeInput,side)
```

MATLAB Code (continued) example2.m

```
%---------------------------- SOLUTION ---------------------------
areaOut = areaSq(shapeInput,side);  %Call the function and store output
                                    %    in "areaOut"

%----------------------------- OUTPUTS ---------------------------
fprintf('OUTPUTS\n')
fprintf('The area of the %s is %s.\n',shapeInput,num2str(areaOut))

%----------------------- FUNCTION DEFINITION -----------------------
function area = areaSq(shapeInput,side)
    if strcmp(shapeInput,'square') %Checking to see if shape is supported
        if side <= 0  %Checking if dimension is valid (positive)
            warnMsg = ['%g is not a valid shape dimension. '...
                       'The positive value of the number (%g) was'...
                       'used to find the area.'];
            warning(warnMsg,side,abs(side))
            side = abs(side);
        end

        area = side^2;  %Finding the area of a square
    else  %If shape is not supported
        area = 'unknown';
        error('This function only supports finding the area of squares.')
    end
end
```

```
Command Window Output                                    Example 2
PURPOSE
To find the area of a square.

INPUTS
The square has input dimension of 6.

OUTPUTS
The area of the square is 36.
```

In the following two Command Window outputs, you can see two lines referenced in each Command Window output. One line is referencing where the problem (warning/error) occurred in the function (actual cause of the problem), and the other one is the line where the function is called. The information given next to the line reference (e.g., "L_7_02_Ex2") will help you identify the location of the warning/error more precisely.

The following Command Window shows the output for the function call `areaSq('circle', 6)`.

```
Command Window Output                                    Example 2
PURPOSE
To find the area of a square.

INPUTS
The circle has input dimension of 6.

Error using L_7_02_Ex2>areaSq (line 36)
This function only supports squares.

Error in L_7_02_Ex2 (line 16)
areaOut = areaSq(shapeInput,side);  %Call the function and store output
```

The following Command Window shows the output for the function call `areaSq('square', -6)`.

```
Command Window Output                                    Example 2
PURPOSE
To find the area of a square.

INPUTS
The square has input dimension of -6.

Warning: -6 is not a valid shape dimension. The positive value of the number
(6) was used to find the area.
> In L_7_02_Ex2>areaSq (line 29)
  In L_7_02_Ex2 (line 16)
OUTPUTS
The area of the square is 36.
```

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Example Usage**
Execute a MATLAB warning	`warning()`	`warning('Message to user')`
Execute a MATLAB error	`error()`	`error('Message to user')`

Multiple Choice Quiz

(1). For the `help` command to work for a function, a comment has to be placed

(a) immediately after the function definition line

(b) on the next line after the function definition line

(c) on the last line in the function

(d) anywhere in the function

(2). You are asked to write a function that finds the area of a triangle by using Hero's formula $A = \sqrt{s(s-a)(s-b)(s-c)}$ where $s = (a+b+c)/2$ and a, b, and c are the lengths of the sides. When writing an error message in the function, a good case to address would be

(a) a, b, and c are positive

(b) $s(s-a)(s-b)(s-c)$ is positive

(c) $s(s-a)(s-b)(s-c)$ is negative

(d) s is not an integer

(3). The `warning()` function accepts inputs of only

(a) static text

(b) text with one variable input

(c) variable inputs

(d) text and/or one or more variable inputs

(4). When an `error()` statement is enumerated and executed in a function, it displays

(a) program name

(b) function name

(c) line number

(d) all of the above

(5). When a `warning()` is triggered,

(a) a custom message is displayed

(b) the program halts

(c) the program pauses until the user continues execution

(d) the program does not show the line number where the warning occurred

Problem Set

(1). Add a description to the function you wrote for Exercise 2 in Lesson 7.1. Give a description of the purpose of the function and a definition of each input and output variable along with what type of data it should contain (e.g., number, string, etc.) and their units when applicable.

(2). Add a description to the function you wrote for Exercise 3 in Lesson 7.1. Give a description of the purpose of the function and a definition of each input and output variable along with what type of data it should contain (e.g., number, string, etc.) and their units when applicable. Add an error message if the user inputs a negative annual percentage rate (APR).

(3). Add a description to the function you wrote for Exercise 6 in Lesson 7.1. Give a description of the purpose of the function and a definition of each input and output variable along with what type of data it should contain (e.g., number, string, etc.) and their units when applicable. Add a warning message if the user inputs a negative income.

(4). Add a description to the function you wrote for Exercise 7 in Lesson 7.1. Give a description of the purpose of the function and a definition of each input and output variable along with what type of data it should contain (e.g., number, string, etc.) and their units when applicable.

Write a MATLAB function that finds the depth to which a metal spherical hollow ball is submerged underwater (see Figure A on the next page), when the density of the metal, the outer radius of the ball, and the thickness of the ball are given. Assume the density of water is 1000 kg/m^3. Test your function three times for three different physically acceptable input choices.

Add a description to the function you write. Give a description of the purpose of the function and a definition of each input and output variable along with what type of data it should contain (e.g., number, string, etc.) and their units when applicable.

Hint: Apply the Archimedes' principle. The volume of the ball underwater for a given depth H is

$$V = \frac{\pi H^2 \left(3r_{\text{out}} - H\right)}{3}$$

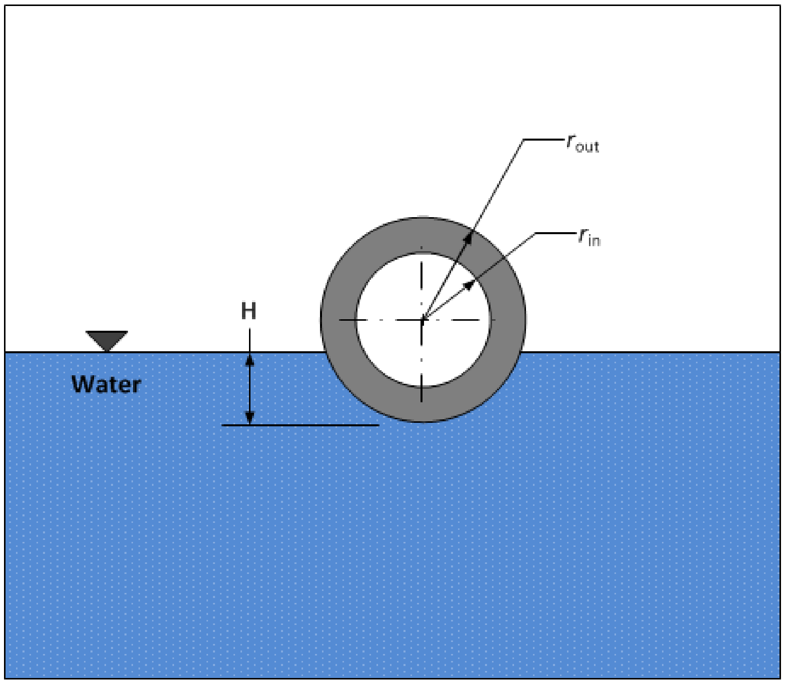


Figure A: Metal ball at equilibrium floating in water.

Module 8: Loops

Lesson 8.1 – while Loops

Learning Objectives

After reading this lesson, you should be able to:

- *differentiate between a definite and an indefinite loop,*
- *write programs using `while` (indefinite) loops with one condition,*
- *write programs using `while` loops with multiple conditions.*

In all the previous lessons, we have considered two basic control structures: sequence and conditional. In this lesson, we introduce you to the control structure of repetition (or “loops”).

What is a loop?

In programming, a loop is a syntax used to describe the action of repeating a block of code (task) more than once. This block of code (task) is commonly referred to as the body of the loop. In Figure 1, we can see two equivalent representations of code. On the left side, the same block of code is repeated multiple times explicitly, while a loop is used on the right side.

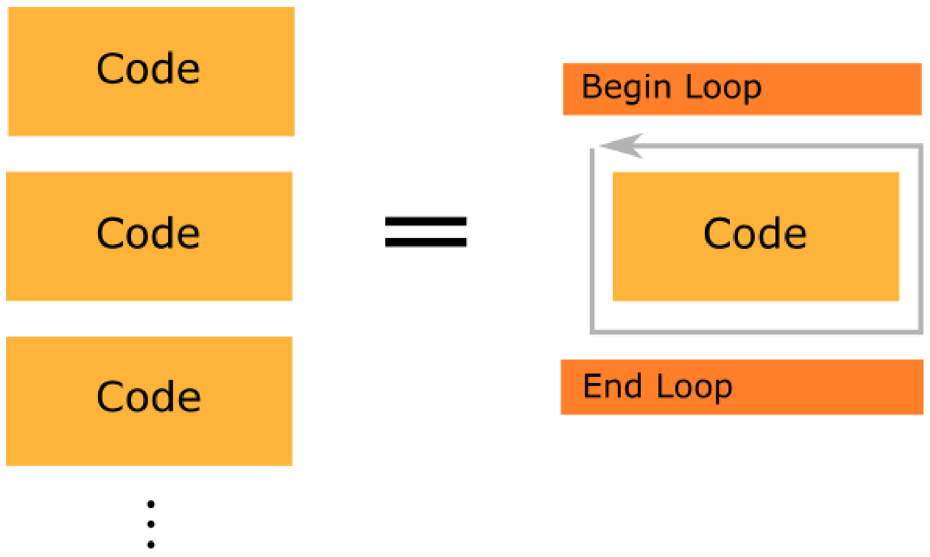


Figure 1: The fundamental structure of loops is to run the same block of code many times.

There are two types of loops: the `for` loop and the `while` loop. The `for` loop will conduct a task a definite number of times because its repetition is controlled by a counter, whereas the while loop will perform a process an indefinite (not to be confused with an infinite) number of times because its repetition is controlled by a logical expression. We will cover `while` loops in this lesson and for loops in Lesson 8.2.

What is a while loop?

The `while` loop conducts an indefinite number of repetitions (loops), where the number of repetitions is controlled by a conditional expression. The `while` loop will continue to conduct repetitions until the conditional expression becomes false. Once the conditional expression is false, MATLAB exits the `while` loop and continues to execute the m-file from the lines below the loop end statement. The four main components of a `while` loop are

1. the statement (`while`),
2. conditional expression,
3. the body of the loop,
4. `end` statement.

The conditional expression(s) used in the `while` loop are the same type of comparisons (for example, >, <=, ~=) and logical operators (for example, ||, &&) used in if statements.

When programming with while loops, one must be careful to avoid an infinite loop. Remember, the loop will continue to run until the conditional expression is false. If you find yourself in an infinite loop (where the conditional expression never becomes false) in MATLAB, simply click inside the Command Window and hit `Ctrl+c` to end the execution of the program. However, if you have pressed "run" multiple times, you will need to repeat the stop command (`Ctrl+c`) multiple times.

Important Note: *Be careful of infinite loops!* An infinite loop is when you write a condition that is *always* true and never becomes false. For example, `while 1 \> 0`.

Example 1

Output the square of all the integers from 3 to 7 in the Command Window. If one wants to write out the square of the integers from 3 to 7, one can write a MATLAB code for it as follows:

```
i = 3;

fprintf(\'Square of %g is %g\', i, i\^2)

i = 4;

fprintf(\'Square of %g is %g\', i, i\^2)

i = 5;

fprintf(\'Square of %g is %g\', i, i\^2)

i = 6;

fprintf(\'Square of %g is %g\', i, i\^2)

i = 7;

fprintf(\'Square of %g is %g\', i, i\^2)
```

As one can see in the above code, the only thing changing in each line is the value of i (also compare with Figure 1). Now take the case where one has to find the squares of numbers from 3 to 100, you will have a lot of code to write. This is a good example of showing the need for a loop.

Solution

Now, we will solve the problem using a `while` loop. Notice that a pseudocode is first made to help identify what variables are changing and what expressions to display. When programming with loops, you may be tempted to jump in with both feet, but you need to clearly identify which variable(s) are changing, and which segments of program are repetitive.

Pseudocode for Example 1:

```
1. Start program, clear window/variables.
2. Set up a loop to find the square of numbers
     Starting number = 3
     Final number = 7
     Increment by 1
     Square each number.
3. Display output
4. End loop when final number is reached
5. End program
```

Remember, all loops must be terminated with an end statement.

In Example 1, the value of `i` is changing. The `while` loop starts with the value of `i` being 3, and then increments the value of `i` by 1 until the loop completes with the value of `i` being 7. There are multiple correct solutions for the while-end loop in Example 1 – one could change the condition of the while statement and the placement of the incrementing line (`i = i + increment`). For example, another correct solution could include the while loop condition as `i \< endNum`. Consider what else would need to be changed to get the same output as shown in the Command Window Output. Doing exercises like this will be helpful when solving more complex problems with loops because it will deepen your understanding of the fundamentals.

Example 2

Using a while loop, write a program that counts to four. The counting should be displayed in the Command Window.

Solution

The iterator, `i`, is what counts for us in the solution. We can name the iterator any valid variable name (other examples for this case are num or count).

It is essential to understand that the condition for the while loop can be *anything* that fits your problem and does not make the while loop infinite. Although the examples we have seen so far have while loops that solely rely on a counter variable for their conditions, there are other common examples that have distinctly different conditions. Some informal examples of this are:

1. Let us say you have a case where an engineer wants to gather sensor data. We might want to write a program using a while loop that stops based on user input.

```
stop = false;

while stop == false

%Code to get sensor reading

end
```

2. In Example 3, we want to calculate values of a function over a specific range: so we use a counter (the independent variable, x) in the condition. However, we might, instead, want to stop the loop when $y(x)$ becomes negative or reaches a specific value. In these instances, we would write our condition with those in mind rather than the counter (iterated variable).

Example 3

Given $y(x) = x^2 - 49$, display the value of y only when y is positive for $x = -10, -9,$ ... $9, 10$.

Solution

Pseudocode for this example:

```
1. Clear Command Window and all workspace variables.
2. Initialize the starting value of x

     starting point is -10

3. Conduct loop repetitions while true.

     number, x less than or equal to 10
```

```
4. Loop body:

      Conduct, $y = x^2 - 49$
      Place logic test: is $y > 0$?
            If true, display $y$
      Increase count by one

5. End loop
```

The Command Window for Example 3 shows that although the while loop is continuing to run for all integer values of x from -10 to 10 as we required; a conditional statement inside of the loop ensures that only positive values of y are displayed in the Command Window.

Example 4

Write a while loop to find the value of x which is updated recursively by

$$\frac{1}{2}\left(x + \frac{9}{x}\right)$$

Use a starting value of $x = 64$ and do the recursion 10 times. Display the last updated value of x as the only output.

Solution

Pseudocode for this example:

```
1. Clear Command Window and all workspace variables.
2. Initialize loop count, starting at 1.
3. Continue loop while true:

      While count is less than or equal to 10

4. Body of loop:

      $x = (1/2) * (x + (9/x))$

5. Display last updated value of $x$
6. End loop
```

The program increases the loop counter, i, by one for each repetition. Once the loop counter, i, is greater than 10, the while loop conditional expression is false and the loop terminates. Observing precisely how loops work from one iteration to the next is essential to being successful in this module.

To give you a background of the above example from a practical point of view, the recursive formula is a way to find the square root of 9. In fact, you can find the square root of any positive real number R by using the recursive formula

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{R}{x_i}\right)$$

What comparisons can I use with a while loop?

The comparisons used with the while loop are the same as those used for conditional statements (if statements). These are shown in Table 1. Just like in conditional statements, you may make more than one comparison in the while loop. You can join each comparison by using the && (AND) and || (OR) operators.

Table 1: Operators to be used for while loop comparison.

Meaning	**Code**
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=
Equal to	==
Not equal to	~=
Boolean Operators	
AND	&&
OR	||

The while loop is an indefinite loop, and hence we need to be careful as to not let it become an infinite loop! For instance, in Example 5, if the series does not converge, you will have yourself an infinite loop. To prevent this from happening, we add a condition to the `while` loop that limits the maximum number of terms added to the series, maxTerms. As soon as the number of terms used, term, becomes greater than the maximum number of terms allowed, maxTerms, the loop condition will not be met, and thus the loop will end.

The value of the exponential function, e^x, can be found by the following series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!}$$

Write a program that uses a while loop to find the value of e^x. Define value of x and stop the loop once the absolute relative approximate error is less than 0.1%. The definition of the absolute relative approximate error is

$$\text{Absolute Relative Approximate Error} = \frac{|\text{Previous Approximation} - \text{ Present Approximation}|}{|\text{Present Approximation}|} \times 100\ \%$$

Example 5

Test your program for $x = 0.75$. Display the final approximation for e^x, the number of terms used, and the last absolute relative approximate error calculated.

Solution

First, one must establish what the inputs are:

1. the number to be evaluated, `x`,
2. the desired absolute relative error (also called pre-specified tolerance), `tolerance`.

Now, we define the outputs as:

1. value of e^x, `exp1`, and,
2. absolute relative approximate error, `ARAE`,
3. number of terms used, `term`.

The series expression for e^x can be rewritten as

$$e^x = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!} + ...$$

and hence in the compact mathematical form as

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Example 5 shows how a `while` loop can be implemented to find the value of a series within a pre-specified tolerance. Because we need to keep adding terms until the pre-specified tolerance is met, the number of terms to be used is not pre-determined, which is why we use an indefinite loop. It should be noted that as one decreases the pre-specified tolerance, more terms may need to be added to achieve the same level of accuracy. Note that we initialize the absolute relative approximate error, `ARAE`, as a number bigger than the pre-specified tolerance, `tolerance`, by adding 1 to it. This is done to get the `while` loop to start the first time around.

Lesson Summary

Task	Syntax	Example Usage
Iterate over a block of code indefinitely	`while`	`a=0; while a<5; a=a+1; disp(a); end`

Multiple Choice Quiz

Problem Set

(1). Write a program using while loop that adds the number 7 to each value of j, as j takes on integer values of 1,2,...,11,12. Output all 12 values to the Command Window.

(2). Write a program using while loop that adds the number 7 to each value of j, as j takes on integer values of 12,11,...,2,1. Output all 12 values to the Command Window.

(3). Using a while loop, write a program that adds together all integers from –20 to 20.

(4). Using a while loop, write a program that outputs $\cos(x)$ values until $\cos(x)$ changes to a negative number. Take values of x from 0 to 2π in increments of 0.1.

(5).Using a while loop, write a program that adds together the elements of any sized vector. Test and run your program using the vector vec =

$$258 - 470 - 9$$

.

(6). Write a MATLAB program that conducts the following summation

$$\text{sum1} = 2 + 3 + 4 + ... + (n + 1)$$

where,

n is the number of terms used.

Use the while loop to perform the summation of the first 16 terms.

(7). Using your knowledge of the while loop and conditional statements, write a MATLAB program that determines the value of the following infinite series

$$f(x) = \frac{1}{2}x + \frac{1}{4}x^2 + \frac{1}{2}x^3 + \frac{1}{4}x^4....$$

There are two program inputs, which are,

1. the value of x, and
2. the number of terms to use.

There is one program output, which is

1. the numeric value of the series.

Your program must work for any set of inputs. You may assume that the value for the number of terms to use will always be entered as a positive whole number.

Test your program for the following set of inputs:

number of terms $= 32$

value of $x = 0.46$

(8).The function, $\cos(x)$ can be calculated by using the following infinite Maclaurin series

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$$

The absolute percentage relative approximate error, $|\varepsilon_a|$ is defined as

$|\varepsilon_a| = \left|\frac{\text{Present Approximation}-\text{Previous Approximation}}{\text{Present Approximation}}\right| \times 100.0$.

Complete the following.

a. Write the pseudocode for a function that finds the approximate value of $\cos(x)$. The function inputs are the argument, x, a pre-specified error tolerance, tol, and a maximum number of terms to use nmax. There are two ways that the loop could end: either it meets the pre-specified tolerance or it uses the maximum number of terms allowed.

b. Write a MATLAB function, myCos, using while loops for calculating $\cos(x)$. The stopping criterion is if a pre-specified tolerance is met or if a specified number of terms are used.

The function inputs are

1. the value at which $\cos(x)$ needs to be calculated, x,
2. pre-specified tolerance, tol,
3. the maximum number of terms allowed, nmax.

The function outputs are

1. the value of $\cos(x)$ when either the maximum number of terms are used or the pre-specified tolerance is met, cosVal,
2. last absolute relative approximate error calculated, absApproxError,
3. the number of terms used, terms, and
4. how the function terminated, howEnded. Assign the integer 1 to howEnded if the pre-specified tolerance is met, and 2 if the maximum number of terms are used.

c. Test your function in part (b) with four different, well-thought-out input variable sets. All four tests are to be made in the same test m-file.

(9). Provided with the following geometric series:

$$S = a + ar + ar^2 + ... + ar^n$$

write a MATLAB program using the while loop to determine the value of S given the inputs a, r, and n.

The program inputs are:

1. the constant, a. (a$\neq$ 0)
2. the value, r (r $\geq$ 0), and
3. term constant, n as n+1 is the number of terms, n $\geq$ 1.

The program output is:

1. The numeric value of S.

Test and run your program for the following combination of a = 3, r = 2.1, and n = 8.

Module 8: Loops

Lesson 8.2 – for Loops

Learning Objectives

After reading this lesson, you should be able to:

- *write programs using for (definite) loops,*
- *use a for loop to plot multiple functions,*
- *decide when to use a for loop vs a while loop.*

What is a for loop?

In this lesson, the discussion is limited to the counter-controlled repetition: that is, the for loop. A `for` loop has the same basic purpose as a `while` loop: to loop, or iterate, over a block of code called the body; that is, to execute the same lines of code over and over until stopped. The `for` loops are defined by defining a loop counter variable, or "iterator". The loop counter variable, or "iterator", keeps track of which repetition the loop is on.

1. The loop counter variable takes the initial value of `startingValue`,
2. The loop counter is checked if it is less than or equal to the `endValue` (the check is greater than or equal to the `endValue` if the increment is a negative number).
3. If so, the loop executes the body of the loop, and the loop counter variable is incremented by the value of increment. Step 2 is repeated, and if the check turns out to be false, the loop is terminated.

As we learned in Lesson 8.1 about `while` loops, this is a variable that changes the value after running each loop iteration. The iterator must have definite bounds (a starting and ending value), and the `for` loop will stop once it reaches the end of this specified bound. In general, this means the `for` loop will execute the same number of times regardless of all other conditions.

For example, in the loop definition for `i = 3:2:9`, the loop counter variable is named i. It will take the initial value of `3` and end at `9` in increments of `2` (four repetitions will be made with values of `i = 3,5,7,9`).

If required, the `startingValue`, increment, and/or `endValue` can be negative.

For example, in the loop definition for `i = 11:-3:2`, the loop counter variable is named `i`. It will take the initial value of `11` and end at `2` in increments of `-3` (four repetitions will be made with values of `i = 11,8,5,2`).

Example 1

Output the square of all the integers from `3` to `7` in the Command Window.

Solution

If one wants to write out the square of the integers from `3` to `7`, one can write a MATLAB code for it as follows:

```
i = 3;

fprintf('Square of %g is %g',i,i^2)

i = 4;

fprintf('Square of %g is %g',i,i^2)

i = 5;

fprintf('Square of %g is %g',i,i^2)

i = 6;

fprintf('Square of %g is %g',i,i^2)

i = 7;

fprintf('Square of %g is %g',i,i^2)
```

Recall from Lesson 8.1, that this is precisely the type of problem that loops are meant for. In this lesson, we will solve the problem using a for loop. Notice that a pseudocode is first made to help identify what variables are changing and what expressions to display.

Pseudocode for Example 1:

```
1. Start program, clear window/variables.
```

2. *Set up a loop to find the square of numbers [Note this can be a for or while loop. Compare with Example 1 of Lesson 8.1.]*

   Starting number = 3

   Final number = 7

   Increment by 1

   Square each number.

3. *Display output*

4. *End loop*

5. *End program*

Remember, all loops must be terminated with an end statement.

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To find the square of integers from 3 to 7
fprintf('To find the square of integers from 3 to 7.\n\n')
```

MATLAB Code (continued) example1.m

```
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
startNum  = 3;
increment = 1;
endNum    = 7;
fprintf(['Use the numbers between %g and %g when counting using ', ...
          'an increment of %g\n\n'],startNum,endNum,increment)

%------------------------- SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
for i = startNum:increment:endNum
    num = i^2;  %Square each sucessive integer
    %Output within the loop because we want to show what is happening in
    %    each loop.
    fprintf('The number is %g. The square is %g.\n',i,num)
end
```

```
Command Window Output                                   Examp
PURPOSE
To find the square of integers from 3 to 7.

INPUTS
Use the numbers between 3 and 7 when counting using an increment of 1

OUTPUTS
The number is 3. The square is 9.
The number is 4. The square is 16.
The number is 5. The square is 25.
The number is 6. The square is 36.
The number is 7. The square is 49.
```

In Example 1, the value of `i` is changing. The `for` loop starts with the value of `i` being `3`, and then increments the value of `i` by `1` until the loop completes with the value of `i` being `7`. It is important to understand that the value of the variable `i` changes and is checked with each loop repetition.

Take a look at some other examples of how the for loop works.

```
for i = 3:1:7
   fprintf('Square of %g is %g',i,i^2)
end
```

The above for loop would print values of the square of 3, 4, 5, 6, 7 as the loop starts with the value of 3, increments by 1 and stops at 7.

```
for i = 3:2:7
   fprintf('Square of %g is %g',i,i^2)
end
```

The above for loop would print values of the square of 3, 5, 7 as the loop starts with the value of 3, increments by 2 and stops at 7.

```
for i = 3:2:10
   fprintf('Square of %g is %g',i,i^2)
end
```

The above for loop would print values of the square of `3, 5, 7, 9` as the loop starts with the value of `3,` increments by `2` and stops at `9`. Although the end limit is `10`, it does not get to that value as the next variable value after `9` would be `11`, which is outside the range.

```
for i = 10:-2:4
```

```
        fprintf('Square of %g is %g',i,i^2)

    end
```

The above for loop would print values of the square of `10, 8, 6, 4`. The expression used here contains a negative increment, hence the order of the outputs. ### Example 2 {-} Output in the Command Window the square of every other integer from `1` to `8`.

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To find the square of integers from 1 to 8
fprintf('To find the square of integers from 1 to 8.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
startNum = 1;
increment = 2;
endNum = 8;
fprintf(['Use the numbers between %g and %g when counting using ', ...
    'an increment of %g.\n\n'],startNum,endNum,increment)

%------------------------- SOLUTION/OUTPUTS --------------------------
fprintf('OUTPUTS\n')
for i = startNum:increment:endNum
    num = i^2;  %Square each sucessive integer
    %Output within the loop because we want to show what is happening in
    %    each loop.
    fprintf('The number is %g. The square is %g.\n',i,num)
end
```

Command Window Output Examp

```
PURPOSE
To find the square of integers from 1 to 8.

INPUTS
Use the numbers between 1 and 8 when counting using an increment of 2

OUTPUTS
The number is 1. The square is 1.
The number is 3. The square is 9.
The number is 5. The square is 25.
The number is 7. The square is 49.
```

In Example 2, the range of values of the loop counter variable i starts at 1 and ends at 8. Why does this for loop stop at i = 7? The answer lies in the increment of the loop counter. This program specifies that the increment is 2, thus MATLAB starts with `i = 1`, followed by `i = 3, 5`, and finally `7`. The next value that i would take on would be 9, but this is greater than the end value of 8. Hence, 7 is the final value of the loop counter variable.

How can I reference vectors inside of a loop?

With loops, the possibilities are endless! Any task that needs to be conducted more than once is an ideal job for a loop. In engineering, mathematics, and science, loops are often used to manipulate arrays, find the value of a series or generate a term of a sequence, solve differential equations numerically, find an integral numerically, and many more tasks.

Example 3 shows how loops are used to conduct a mathematical operation: summing a vector of numbers. To do this, we will combine two important concepts we have learned. We have referenced specific elements of a vector before (e.g., vec(1)), but to find the sum of a vector of any size, we need to use a loop to "loop through" all the elements in the vector. Therefore, vec(1)+vec(2)+vec(3)+... becomes vecSum = vecSum + vec(i).

Example 3

Given the following vector data =

$$2, 7, 4.5, 2.3, 6, 5$$

, find the sum of all the elements by using a for loop.

Output the value of the summation by using fprintf(). Be sure to develop a loop that will work for any size vector.

Solution

Pseudocode for Example 3:

1. *Clear Command Window and all variables.*
2. *Prompt user to input data set, as a vector.*
3. *Use length() to find the number of elements.*
4. *Initialize the loop counter variable used for the summation.*
5. *Start loop.*
6. *Index from 1 to final element number.*
7. *Add each element together in the body of the loop.*
8. *End loop.*
9. *Display results.*

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To sum all the numbers in a vector
fprintf('To sum all the numbers in a vector.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
data = [2 7 4.5 2.3 6 5];
disp('The input data is:')
disp(data)

%------------------------------ SOLUTION ----------------------------
len = length(data);  %Number of elements in the vector
vecSum = 0;          %Starting point for summation
for i = 1:1:len
    vecSum = vecSum + data(i);
end

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The sum of the vector is %g\n',vecSum)
```

Command Window Output Exam

```
PURPOSE
To sum all the numbers in a vector.

INPUTS
The input data is:
    2.0000    7.0000    4.5000    2.3000    6.0000    5.0000

OUTPUTS
The sum of the vector is 26.8
```

In the m-file, note that the variable named dataSum is equal to zero before starting the loop. Why does the m-file require that dataSum be equal to zero before even starting the loop? Well, think back to the fundamentals of programming. When MATLAB sees a variable name, it asks what its value is. Therefore, when

MATLAB evaluates the line dataSum = dataSum + 1 for the first time (i=1), the dataSum on the right side would be undefined (if we had not initialized it). This, of course, will give an error. That is why we initialize dataSum = 0 before the loop starts: so MATLAB will know the value of dataSum on the first iteration of the loop. Once the loop finishes the first repetition, a new value of dataSum overwrites the previous value stored in dataSum, and this continues till the loop has run its course.

Example 4

***Background*:** A legend says that King Shriham of India wanted to reward his grand minister, Ben, for inventing the game of chess. There are 64 squares on a chessboard. When asked what reward he wanted, Ben asked for one grain of rice on the first square of the board, two on the second square of the board, four on the third square of the board, eight on the fourth square of the board, and so on until all squares are covered. That is, he was doubling the number of grains on each successive square of the board. Although Ben's request looks less than modest, King Shriham quickly found that the amount of rice would be many times more than his country could ever produce. Can you find out how much rice Ben was asking for?

***Specifications*:** Write a program that uses a loop to calculate the total amount of rice on the first n squares of a chessboard given the above scenario. Hint: if four squares on the board are used, then you have eight grains on the last square, and the total grains of rice is 1 + 2 + 4 + 8 = 15.

The program input is the number, n (n $\leq$ 64), of squares used.

The outputs are,

1. the number of grains on the last square used that reads: The total number of grains on square ??? is ???".

2. the number of grains of rice on the board that reads: The total number of the grains is ????.

MATLAB Code examp

```
clc
clear

%----------------------------- PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To solve a mathematical puzzle
fprintf('To solve a mathematical puzzle.\n\n')

%----------------------------- INPUTS -------------------------------
fprintf('INPUTS\n')
n = 64;  %Number of squares filled
fprintf('The number of squares used is %g.\n\n',n)
```

MATLAB Code (continued) examp

```
%----------------------------- SOLUTION ----------------------------
totalGrains = 0;     %Starting point for summation
for i = 1:1:n
    currentGrains = 2^(i-1);  %Each square has 2^(i-1) grains
    totalGrains = totalGrains + currentGrains;
end

%----------------------------- OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('Number of grains on the last square is %g.\n',currentGrains)
fprintf('Total number of grains on the board is %g.\n',totalGrains)
```

Command Window Output Exam

```
PURPOSE
To solve a mathematical puzzle.

INPUTS
The number of squares used is 64.

OUTPUTS
Number of grains on the last square is 9.22337e+18.
Total number of grains on the board is 1.84467e+19.
```

The grains on the current square is assigned to the variable currentGrains. The variable currentGrains is added to the totalGrains to keep track of the grains on the board.

Do I have to use the loop counter variable in the body of the loop?

No, you do not have to use the loop counter variable in a for loop. The purpose of the loop counter variable and its assignment in the for loop is to control the number of times the loop will repeat itself.

One can see that in the solution in Example 5, the loop counter variable j is not used in the body of the for loop. In this case, the loop counter variable is solely used as a counter, to keep track of how many repetitions that this loop has made.

Example 5

You can find the square-root, x, of any positive real number, R, by using the recursive formula given as

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{R}{x_i}\right)$$

Write a for loop to find the value of x for $R = 9$, which is updated recursively by $\frac{1}{2}\left(x + \frac{9}{x}\right)$. Use a starting value of `x = 64` and do the recursion 10 times.

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To implement a recursive mathematical formula
fprintf('To implement a recursive mathematical formula.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
x         = 64;  %Starting value of x
numReps   = 10;  %Number of loops
fprintf('The starting value of x is %g.\n',x)
fprintf('x will be updated recursively %g times.\n\n',numReps)

%------------------------------ SOLUTION -----------------------------
for j = 1:1:numReps
    x = (1/2)*(x + (9/x));  %Implement recursive formula that updates
                            %    value of x for each loop
end

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The updated value of x is %g.\n',x)
```

Command Window Output Exam

```
PURPOSE
To implement a recursive mathematical formula.

INPUTS
The starting value of x is 64.
x will be updated recursively 10 times.

OUTPUTS
The updated value of x is 3.
```

To review, all for loops require a loop counter variable assigned to a vector of `startingValue:increment:endValue`, a body of code, and an end statement. The loop counter variable does not need to be used inside the loop (recall Example 5) because it is just a counter to keep track of repetitions.

When do I use a for loop vs. a while loop?

There are no absolute rules for when to use a `for` loop over a `while` loop or vice versa. In almost every case, you can accomplish the task with either a `for` or a `while` loop (this is because you can use conditions to change the behavior of a `for` loop into a while-like loop). The difference is the efficiency or elegance of your solution. There is generally no right answer, so you should focus on learning how each one works rather than when to use each one.

Examples 6 and 7 are given purely to demonstrate the similarities between using a `for` and `while` loop to solve the same simple problem. As mentioned above and in later lessons, there are cases where one is easier and more appropriate to implement than the other.

Example 6

Write a program that sums all the numbers between `1` and `n` using a for loop. Display the final sum in the Command Window. Use `n = 10` as a test case for the program.

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To sum all the numbers between 1 and n
fprintf('To sum all the numbers between 1 and n.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
n = 10;             %Defining the ending value of the iterator
fprintf('Sum the numbers between 1 and %g.\n\n',n)

%------------------------------ SOLUTION ------------------------------
sumOfNumbers = 0;  %Defining variable that will store the sum of all
                   %  values of the iterator
for i = 1:1:n                           %Defining the for loop w
increment (step size) of 1
    sumOfNumbers = sumOfNumbers + i;    %Summing all iterator values
                                    %    (1 + 2 + 3 + 4 + … n)
end

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The sum of the numbers between 1 and %g is %g.\n',n,sumOfNuml
fprintf('The answer using the MATLAB sum function is %g.\n',sum(1:n))
```

MATLAB Code example6.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To sum all the numbers between 1 and n
fprintf('To sum all the numbers between 1 and n.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
n = 10;             %Defining the ending value of the iterator
fprintf('Sum the numbers between 1 and %g.\n\n',n)

%------------------------------ SOLUTION ----------------------------
sumOfNumbers = 0;  %Defining variable that will store the sum of all
                   %  values of the iterator
for i  =  1:1:n                                %Defining  the  for  loop  with  an
increment (step size) of 1
    sumOfNumbers = sumOfNumbers + i;   %Summing all iterator values
                                   %   (1 + 2 + 3 + 4 + … n)
end

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The sum of the numbers between 1 and %g is %g.\n',n,sumOfNumbers)
fprintf('The answer using the MATLAB sum function is %g.\n',sum(1:n))
```

Remember, it would be wrong to name "`sumOfNumber`" as "`sum`" since "`sum`" is a predefined MATLAB function. If it was named "`sum`", the actual function "`sum()`" would be redefined, and would not work in the final `fprintf()` function.

Now, we can solve the same problem using a `while` loop. We achieve the same answer in both Examples 6 and 7.

Example 7

Write a program that sums all the numbers between `1` and `n` using a `while` loop. Display the final sum in the Command Window. Use `n = 10` as a test case for the program.

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To sum all the numbers between 1 and n
fprintf('To sum all the numbers between 1 and n.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
n = 10;        %Defining the ending value of the iterator
fprintf('Sum the numbers between 1 and %g.\n\n',n)

%------------------------------ SOLUTION -----------------------------
sumOfNumbers = 0;  %Variable that will store the sum of all iterator
i            = 0;  %Initialize i to be 0 since we add 1 before summin
while i < n             %Defining the while loop and its condition
       i = i + 1;       %Adding to/Increment the iterator by 1
       sumOfNumbers = sumOfNumbers + i;  %Summing all iterator values
                                         %   (1 + 2 + 3 + 4 + … n)
end

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The sum of the numbers between 1 and %g is %g.\n',n,sumOfNum
```

MATLAB Code example7.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To sum all the numbers between 1 and n
fprintf('To sum all the numbers between 1 and n.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
n = 10;         %Defining the ending value of the iterator
fprintf('Sum the numbers between 1 and %g.\n\n',n)

%------------------------------ SOLUTION -----------------------------
sumOfNumbers = 0;  %Variable that will store the sum of all iterator
i            = 0;  %Initialize i to be 0 since we add 1 before summing
while i < n            %Defining the while loop and its condition
       i = i + 1;      %Adding to/Increment the iterator by 1
       sumOfNumbers = sumOfNumbers + i;  %Summing all iterator values
                                         %    (1 + 2 + 3 + 4 + … n)
end

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The sum of the numbers between 1 and %g is %g.\n',n,sumOfNumbers)
```

Lesson Summary

Task	Syntax	Example Usage
Iterate over a block of code definitely	`for`	`for a = 1:5; disp(a); end`

Multiple Choice Quiz

Problem Set

(1). Write a program using a `for` loop that adds the number 7 to each value of `j`, as `j` takes on integer values of 1,2,...,11,12. Output all 12 values to the Command Window.

(2). Write a program using a `for` loop that adds the number 7 to each value of j, as j takes on integer values of 12,11,...,2,1. Output all 12 values to the Command Window.

(3). Using loops, write a program that adds the number 7 to each value of `j`, as `j` takes on integer values of 1, 3, ...,9, 11. Output all values to the Command Window.

(4). A function $f(x)$ is calculated by using the following infinite series,

$$f(x) = \frac{1}{3} + x + x^2 + x^3 + x^4 + ...$$

Complete the following: (a) Use the `syms` command with `x` as the symbolic character to show the series in the Command Window. The only program input is the number of terms to use, `n`.

(b) Calculate the value of $f(x)$ using an x value of 1.24 and the first 30 terms of the infinite series.

(5). The function, $cos(x)$ can be calculated by using the following infinite Maclaurin series

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$$

The absolute percentage relative approximate error $|\varepsilon_a|$ is defined as

$$|\epsilon_{\mathrm{a}}| = \left|\frac{\text{Present Approximation} - \text{Previous Approximation}}{\text{Present Approximation}}\right| \times 100.0$$

(a) Find $cos(0.5)$ using five terms of the series.

(b) Find $cos(0.5)$ using six terms of the series. Find the absolute percentage relative approximate error at the end of using the six terms.

(6).Find the average and standard deviation of a given vector of numbers. The average of numbers $(x_1, x_2, x_3, ..., x_n)$ is given by

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

and the standard deviation of the numbers is given as

$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}}$.

Write a program using `for` loops to find the average and the standard deviation of given numbers in a vector.

(7).The function $f(x)$ is calculated by using the following infinite series,

$$f(x) = \frac{x^2}{17} + \frac{x^3}{3} + \frac{x^4}{3} + \frac{x^5}{3} + ...$$

Complete the following:

(a) Write a pseudo code for a MATLAB function that finds the value of $f(x)$, given the number of terms to use, n and the value, x.

(b) Write a MATLAB function my_fun using `for` loops for calculating $f(x)$. Use n terms to calculate $f(x)$ at a given value of x.

```
function fn=my_fun(x,n)
```

(c) Test the function in a separate m-file for four different cases of input variables:

i) $x = 0.25$, $n = 4$

ii) $x = 0.25$, $n = 61$

iii) $x = 3.00$, $n = 1$

iv) $x = 3.00$, $n = 2$.

(8). Using your knowledge of `for` loops and/or conditional statements, write a MATLAB program that outputs the factorial of any positive input integer `N`. Recall that the factorial of a number is found as $n! = n \times (n-1) \times ... \times 2 \times 1$.

For example,

$$\begin{aligned} 5! &= 5 \times 4 \times 3 \times 2 \times 1 \\ &= 120 \end{aligned}$$

Also, recall that 0! is explicitly defined as equal to 1. Provide an error message in the Command Window if the input number is negative or not an integer. You should not use the factorial() or similar function to complete this problem.

Test run your program with two sets of inputs. First, test run your program with the following input,

```
n = 8
```

where the output would be

```
8! is 40320
```

Then, test run your program with the following input,

```
n = -12
```

where the output would be

```
Error -- The input is negative
```

Lastly, test run your program with the following input

```
N = 12.3
```

where the output would be

```
Error -- The input is not an integer
```

(9).A prime number is a positive whole number which has exactly two distinct number divisors. In other words, a prime number is only divisible by itself and 1. For example, 2, 3, 5, 7, 11, 13 are prime numbers. Note that the number 1 is not a prime number. Use your knowledge of `for` loops and/or conditional statements to write a MATLAB program that determines if an input positive whole number is a prime number. The program input is any positive whole number, `n` ($n \geq 1$), and the program output is whether or not the inputted whole number is a prime number.

Test your program for the number 97, which is a prime number.

Module 8: Loops

Lesson 8.3 – break and continue Commands

After reading this lesson, you should be able to:

- *use the `continue` command in loops,*
- *use the `break` command in loops,*
- *decide between the use of the `continue` and `break` commands.*

What are the `break` and `continue` commands?

The `break` and `continue` commands are a way to manipulate both `while` and `for` loops, while they are running. Given this, both of these commands should always be placed inside conditional statements. This fact will become clear after reviewing the examples in this lesson.

Neither command, once it has been called, allows the current loop iteration to complete (i.e., the rest of the loop body is not going to be executed for that iteration). The commands run exactly where they are placed in the code: not at the end of the loop iteration.

How does the `break` command work in MATLAB?

The `break` command in MATLAB is used to break out of a `for` or `while` loop; that is, it terminates the execution of the loop.

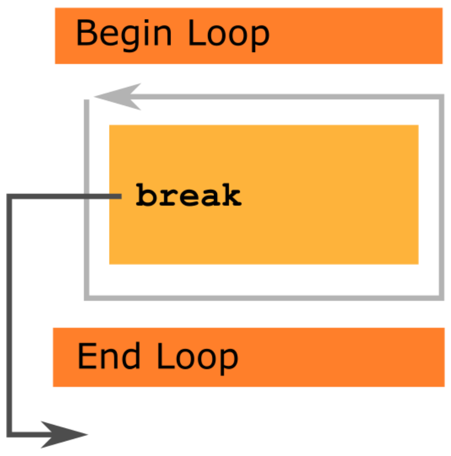


Figure 1: Shows the path of execution when the `break` command executes.

The `break` command should always be used in conjunction with a conditional statement. Once the condition is true, the `break` command is active and the loop breaks. After the `break` command has terminated the loop, the segments of body code below the `break` command will not run. The m-file will continue to run as normal below the loop `end` statement. This process is visually summarized in Figure 1.

Example 1

Write a program that calculates the values of $k^2 - 50$ for all integers in $[-10, 10]$ domain, but only until $k^2 - 50$ becomes negative. Do not continue calculating values after a negative result has been found.

Solution

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To evaluate a function over a given domain until it becomes negative
fprintf(['To evaluate a function over a given domain until it\n',...
         'becomes negative.\n\n'])

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
kDomain = 10;
fprintf('Check the function over the domain [-%g,%g].\n\n',...
    kDomain,kDomain)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
for k = -kDomain:1:kDomain
    val = k^2 - 50;
    if val < 0  %Checking whether k^2-50 is negative
        break  %break is inside of a conditional statement
    end
    fprintf('k = %g  val = %g\n',k,val)
end
```

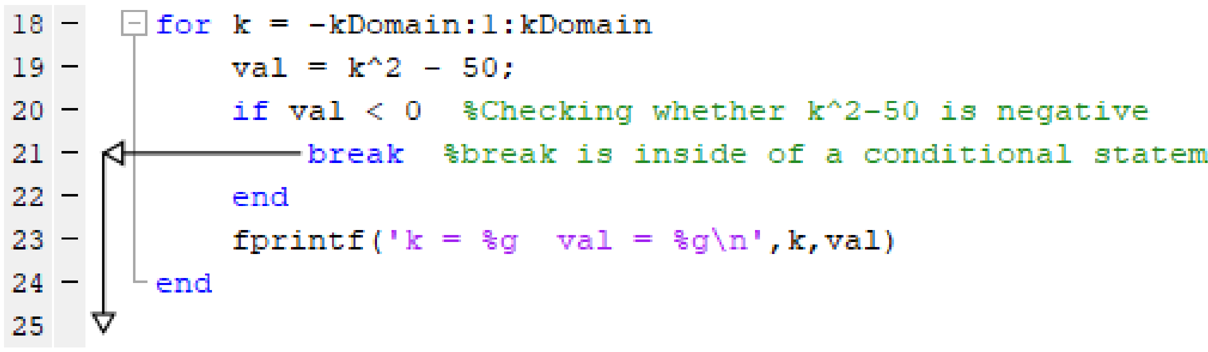
Command Window Output Example 1

```
PURPOSE
To evaluate a function over a given domain until it
becomes negative.

INPUTS
Check the function over the domain [-10,10].
```

Command Window Output (continued) Example 1

```
OUTPUTS
k = -10  val = 50
k = -9  val = 31
k = -8  val = 14
```

Figure 2 shows the program flow when/if the `break` command is executed (line 21). Note the current iteration of the loop does not finish (the message on line 23 is not displayed), and the loop exits immediately. MATLAB then continues to execute the code after line 24 normally.



```
for k = -kDomain:1:kDomain
    val = k^2 - 50;
    if val < 0  %Checking whether k^2-50 is negative
        break  %break is inside of a conditional statem
    end
    fprintf('k = %g  val = %g\n',k,val)
end
```

Figure 2: Programming flow once the `break` command executes.

How does the continue command work in MATLAB?

The `continue` command in MATLAB is used to conditionally pass control to the next repetition in loops (both `for` and `while` loops). Like the break command, the `continue` command should be used with a conditional statement. When the conditional statement is true, the `continue` command initiates the next loop cycle, regardless of the code in the body of the loop. The lines of code below the `continue` command are not run until the `continue` command is inactive.

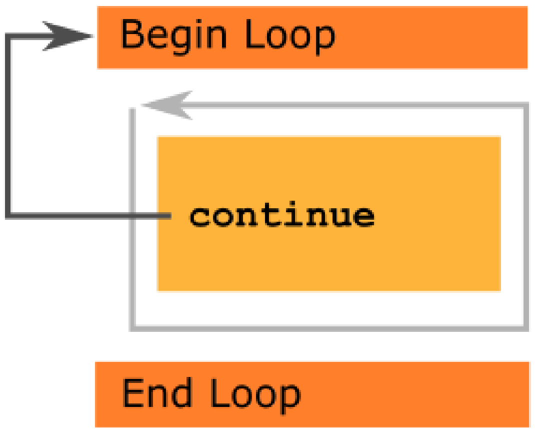


Figure 3: Shows the path of execution when the `continue` command executes.

Example 2

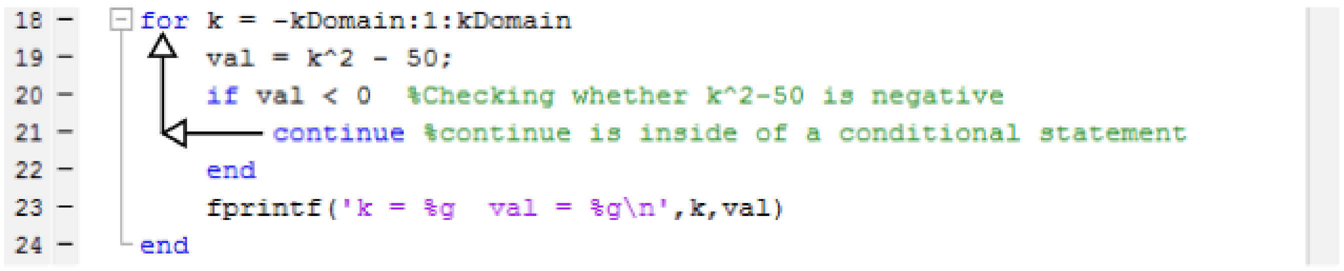
You are asked to calculate and print the values of $k^2 - 50$ for all integers in $[-10, 10]$ domain, but only if $k^2 - 50$ is positive.

Solution

```
MATLAB Code                                                    example2.m
clc
clear

%------------------------------ PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To evaluate a function over a given domain until it becomes negative
fprintf(['To evaluate a function over a given domain while it is ',...
          'positive.\n\n'])

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
kDomain = 10;
fprintf('Check the function over the domain [-%g,%g].\n\n',...
    kDomain,kDomain)

%-------------------------- SOLUTION/OUTPUTS ------------------------
fprintf('OUTPUTS\n')
for k = -kDomain:1:kDomain
    val = k^2 - 50;
    if val < 0  %Checking whether k^2-50 is negative
        continue %continue is inside of a conditional statement
    end
    fprintf('k = %g  val = %g\n',k,val)
end
```

```
Command Window Output                                          Example 2
PURPOSE
To evaluate a function over a given domain while it is positive.

INPUTS
Check the function over the domain [-10,10].

OUTPUTS
k = -10  val = 50
k = -9  val = 31
k = -8  val = 14
k = 8  val = 14
k = 9  val = 31
k = 10  val = 50
```

Whenever the `continue` command becomes active, the execution jumps to the `for` statement of the `for` loop, where k gets incremented.



```
for k = -kDomain:1:kDomain
    val = k^2 - 50;
    if val < 0  %Checking whether k^2-50 is negative
        continue %continue is inside of a conditional statement
    end
    fprintf('k = %g  val = %g\n',k,val)
end
```

Figure 4: Program flow when `continue` command executes.

In Figure 4, we can see that each time the `continue` command executes (line 21), the program immediately (without executing the rest of the loop body) continues to the next iteration of the loop. This means that the message we display with `fprintf()` (line 23) will only display (execute) for positive calculated numbers.

Lesson Summary

Task	Syntax	Example Usage
Exit a loop before it finishes.	`break`	`break`
Continue to the next iteration of a loop before the current iteration finishes.	`continue`	`continue`

Multiple Choice Quiz

Problem Set

(1).Use `for` loop(s) to write a MATLAB program that displays the value of r and w, where

$$w = -7r^2 + 3r + 25$$

for r in the domain [0,40] in steps of 2 units until w turns negative.

(2). Use for loop(s) to write a MATLAB program that outputs all the negative values of $7\sin(x)$ in the x domain of $[0, 3\pi]$. Use an interval of $\frac{\pi}{10}$.

(3). Repeat Exercise 2 with a `while` loop.

(4). Using a `for` loop, write a MATLAB function that finds the voltage (V) measured across a resistor of resistance (R) when a variable current (i) is applied to the resistor. The current is a function of time and is given by

$$i(t) = 2t^2 - 3t$$

where i is measured in amperes and t in seconds. The voltage across the resistor is given by

$$V = iR.$$

The resistor (R) has a constant value of 120 Ω. Find the value of the voltage in the t domain of [0, 20] in increments of 0.1. End finding the voltage when the voltage reaches 4V or more. Output the final value of the voltage and the time this value is reached.

(5). Write a MATLAB program that prompts the user to enter a vector of their choosing and storing the vector as, `userVec`. Via a for loop, the vector is entered one element at a time by the user through an `input()` prompt until the user enters over to denote the end of the vector. The program should stop prompting and display the entire vector in the Command Window.

Hint: You will have to use the vector element notation to complete this problem.

(6). Repeat Exercise 5 using the `while` loop.

Module 8: Loops

Lesson 8.4 – Nested Loops

Lesson Objectives

After reading this lesson, you should be able to:

- *apply knowledge of loop fundamentals to create nested loops,*
- *write programs using nested loops,*
- *use programming flags in loops.*

What is a nested loop?

Nested loops are just what they sound like: a loop within a loop. There are no programming limitations on how many nested loops you can create. By definition, you need a minimum of two loops to have nested loops: one parent and one nested loop. For example, a `for` loop nested inside another `for` loop as seen in Figure 1.

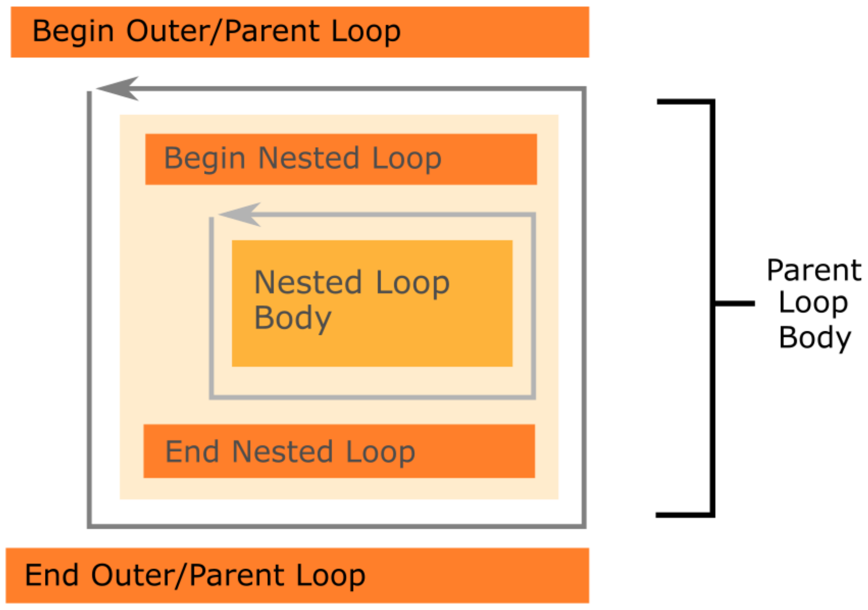


Figure 1: Depicts nested loops where one nested (or inner) loop is contained inside the body of the parent (or outer) loop.

How do nested loops work?

We will go into more examples of nested loop applications in the following lessons. For now, imagine you want to display each element of a vector individually. You will need one `for` loop to "look" through all of those elements. Now, suppose you want to display all the elements of a matrix. Remember, you can think of a matrix as a collection of vectors. You will need to run a `for` loop for each vector contained in the matrix! That is, you will need one `for` loop for each column of the matrix and one `for` loop for each row of the matrix. So, the block of code you want to repeat *contains* a loop.

Example 1

Show each element of any given matrix one element at a time in the Command Window. Test the program with the matrix A.

$$A = \begin{bmatrix} 7 & 11 \\ 13 & 19 \\ 23 & 31 \end{bmatrix}$$

Solution

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To display every element of a matrix one at a time
fprintf('To display every element of a matrix one at a time.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
A = [7 11; 13 19; 23 31]  %Generating an arbitrary square matrix

%------------------------- SOLUTION/OUTPUTS --------------------------
fprintf('OUTPUTS\n')
for i = 1:3
    %Start block of code for i loop which iterates through each row
    for j = 1:2
        %Start block of code for j loop which iterates through each column
        element = A(i,j);
        fprintf('A(%g,%g) = %g\n',i,j,element)
        %End block of code for j loop
    end
    %End block of code for i loop
end
```

Command Window Output Exam

```
PURPOSE
To display every element of a matrix one at a time.

INPUTS

A =
     7    11
    13    19
    23    31

OUTPUTS
A(1,1) = 7
A(1,2) = 11
A(2,1) = 13
A(2,2) = 19
A(3,1) = 23
A(3,2) = 31
```

How do loop mechanics apply to nested loops?

Loop mechanics refers to how loops function fundamentally. For example, how a loop runs a block of code multiple times or how variables are referenced from the previous iteration values. The code will run the entire nested loop before going to the next iteration of the parent loop. You can see this general property of loops exhibited in Example 2. Notice the outer loop only iterates after its block of code executes, which includes the inner loop.

Important Note: The code will run the entire nested loop before going to the next iteration of the parent loop.

Example 2

Demonstrate the mechanics of nested loops by displaying the values of the loop iterators each time either loop iterates.

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To demonstrate the function of nested loops
fprintf('To demonstrate the function of nested loops.\n\n')
```

MATLAB Code (continued) example2.m

```
%------------------------- SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS')
for outer = 1:2
    fprintf('\n******OUTER LOOP RAN******\n')
    for inner = 1:3
        %Remember, the "%.0f" in the fprintf string is telling MATLAB to
        %output 0 decimals (a whole number) from that variable
        fprintf('Outer loop = %.0f and the inner loop = %.0f\n',outer,inner)
    end
end
```

Command Window Output Example 2

```
PURPOSE
To demonstrate the function of nested loops.

OUTPUTS
******OUTER LOOP RAN******
Outer loop = 1 and the inner loop = 1
Outer loop = 1 and the inner loop = 2
Outer loop = 1 and the inner loop = 3

******OUTER LOOP RAN******
Outer loop = 2 and the inner loop = 1
Outer loop = 2 and the inner loop = 2
Outer loop = 2 and the inner loop = 3
```

We can see from both the solution and its result (in the Command Window output) that the program must complete a total of $2 \times 3 = 6$ repetitions.

You can mix and match your nested loops. For example, you can do `for-for`, `while-for`, `for-while`, etc. It just depends on what you need in your solution to a problem. Of course, the general principle of the inner loop being complete before the outer loop iterates remains true as demonstrated in Example 4.

What is a “flag”?

A programming trick often used in loops (especially nested loops) is something called flags. Flags are not a special data type; they are simply a way of signaling in the program. It is kind of like shooting up a flare or raising a physical flag on a mailbox that another part of the program “sees”. We will use flags in Examples 3, 4, and 5.

Typically, one would include the phrase “flag” in the variable name to differentiate it from other variables. It is also good practice to write the different states of the variable when it is first defined. For example, we could define a variable named `stopFlag` and write a comment stating “true means the operation should end”.

When should I use programming flags?

A flag is typically a numeric or Boolean value (Example 3 uses a Boolean flag). If you need a refresher on Boolean, review Lesson 5.1.

Important Note: You can name the flag variable anything just like any other variable.

Example 3

Search any given vector to find the first instance of a given number (if it contains the number). Display a message in the Command Window the first instance that value is found and where its location is within the vector. Use the vector `vec = [1 3 2 9 4 10 4]` and look for the number 4 to test the program.

Solution

MATLAB Code Example3.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To search for the first instance of a number in a vector
fprintf('To search for the first instance of a number in a vector.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
boolFlag = false;  %Initializing flag as with Boolean value. “true” means
                   %    we found the value we are searching for.
vec = [1 3 2 9 4 10 4];  %Creating test vector to search
val = 4;                 %Defining value to search the vector for
fprintf('Determine if the number %g exists in the vector.\n\n',val)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
i = 1;  %Initializing iterator

%Use while loop to go *until* value is found or end of vec is reached
while boolFlag == false && i <= length(vec)
    if val == vec(i)  %if value is found, raise flag
        fprintf('The value %g was found first at element %.0f.\n',...
                  vec(i),i)
        boolFlag = true;     %Raising the flag
    end
    i = i + 1;  %Incrementing loop iterator
end
```

Command Window Output Example 3

```
PURPOSE
To search for the first instance of a number in a vector.

INPUTS
Determine if the number 4 exists in the vector.

OUTPUTS
The value 4 was found first at element 5.
```

Note that this solution stops if one instance of the desired value is found. You can see this since there are two instances of 4 in the given vector, but only one is displayed. The solution can be modified to find all the instances of the desired value in an array.

In Examples 3, 4, and 5, Boolean variables are used as programming flags. They are binary cases that can only either be true or false. This is the most common use case in programming as it makes things very clear at a glance (e.g., `isEmpty = false`, so we know the thing is not empty). However, there are some cases that have more than two states we would like to represent. In these cases, we can use numbers or strings as the values for the flag.

You are tasked with developing a program in MATLAB that generates six random *unique* integers in the 1-53 range and stores each integer in a vector called `luckySix`. Store each integer in the order in which it was picked (i.e., the first number generated is stored as the first element of vector `luckySix` and so on). The program must display these *unique* integers in the Command Window.

Example 4

Solution

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To generate six unique random integers
fprintf('To generate six unique random integers.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
xLow = 1;
xHigh = 53;
n = 6;
fprintf('Generate %g random integers from %g to %g.\n\n',n,xLow,xHig

%------------------------------ SOLUTION -----------------------------
%Generating a random integer between xLow and xHigh (inclusive)
luckySix(1) = floor(xLow + rand*(xHigh - xLow + 1));
i = 2;  %Initialize at 2 since the first value will always be unique
```

MATLAB Code (continued) example4.m

```
while i <= n
    %Generate and store a candidate for a unique integer
    luckySix(i) = floor(xLow + rand*(xHigh - xLow + 1));
    uniqueFlag = true;  %Initializing flag value

    %Checking if numbers are unique
    for j = 1:length(luckySix)-1  %Note: length(luckySix) == i
        %Compare candidate with each value in luckySix
        if luckySix(i) == luckySix(j)
            uniqueFlag = false;  %Setting flag to 1 if integer is unique
        end
    end

    %If integer is unique, move on to the next candidate
    if uniqueFlag == true  %"== true" is for clarity, but it is redundant
        i = i + 1;
    end
end

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The unique "Lucky Six" integers are:\n')
disp(luckySix)
```

Command Window Output Example 4

```
PURPOSE
To generate six unique random integers.

INPUTS
Generate 6 random integers from 1 to 53.

OUTPUTS
The unique "Lucky Six" integers are:
     2    46    50    36    41    40
```

One can see that two loops are required to complete this example. In this case, the inside `for` loop checks the current random integer against all other previous chosen integers. The `while` loop is used because we do not know the exact number of repetitions it would take to pick six unique integers.

Since this code relies on the `rand()` function, you should expect to see different results from those shown below.

How can I use `break` and `continue` in nested loops?

When using `break` and `continue` commands in nested loops (i.e., a loop within a loop), the commands only apply inside of loop it is placed in. Calling `break` inside a nested loop will exit that nested loop; however, it will not exit the parent loop. Example 5 shows how you can exit both loops using a flag. Note that in Example 5, the `break` command appears twice: once to exit the nested loop and once to exit the parent loop.

Important Note: break and continue apply only to the *first* loop that contains them.

Make sure you review the lesson on `break` and `continue` commands (Lesson 8.3) before trying this example.

Example 5

Demonstrate how to use the `break` command inside of nested loops. Once the `break` command has executed within the inner loop, the program should also break out of the outer loop.

Solution

MATLAB Code example4.m

```matlab
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To generate six unique random integers
fprintf('To generate six unique random integers.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
xLow = 1;
xHigh = 53;
n = 6;
fprintf('Generate %g random integers from %g to %g.\n\n',n,xLow,xHigh)

%------------------------------ SOLUTION ----------------------------
%Generating a random integer between xLow and xHigh (inclusive)
luckySix(1) = floor(xLow + rand*(xHigh - xLow + 1));
i = 2;  %Initialize at 2 since the first value will always be unique
```

```
MATLAB Code (continued)                                          example4.m
while i <= n
    %Generate and store a candidate for a unique integer
    luckySix(i) = floor(xLow + rand*(xHigh - xLow + 1));
    uniqueFlag = true;  %Initializing flag value

    %Checking if numbers are unique
    for j = 1:length(luckySix)-1  %Note: length(luckySix) == i
        %Compare candidate with each value in luckySix
        if luckySix(i) == luckySix(j)
            uniqueFlag = false;  %Setting flag to 1 if integer is unique
        end
    end

    %If integer is unique, move on to the next candidate
    if uniqueFlag == true  %"== true" is for clarity, but it is redundant
        i = i + 1;
    end
end

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The unique "Lucky Six" integers are:\n')
disp(luckySix)
```

As seen in Example 5, we need to use a second `break` command to break out of both loops after our condition occurs. Try removing the second `break` command and see what happens!

To make the code a little cleaner, we can remove the explicit check of "true" or "false" with Boolean flags (variables). In Example 5 above, if `stopFlag` functions the same way as if `stopFlag == true`.

Multiple Choice Quiz

(1). The data type of a flag variable is

(a) `char`

(b) `double`

(c) `logical`

(d) any of the above

(2). The purpose of a programming flag is to

(a) be a signal in a program

(b) make a conditional decision

(c) repeat a body of code

(d) find a number in a loop

(3). Complete the code to generate a random integer between 5 and 14.
`number = floor(rand*(____________))+5`

(a) `14 - 5 + 1`

(b) `14 - 5`

(c) `5 - 1`

(d) `14 - 0`

(4). The Command Window output of the following program is

```
clc
clear
sum1 = 0;
for i = 1:1:3
    for j = 1:1:3
        sum1 = sum1 + 4;
    end
end
sum1
```

(a) `sum1 = 0`

(b) `sum1 = 4`

(c) `sum1 = 12`

(d) `sum1 = 36`

(5). The Command Window output of the following program is

```
clc
clear
sum1 = 1;
for i = 1:1:2
    for j = 1:1:3
        sum1 = sum1*2;
    end
end
sum1
```

(a) `sum1 = 18`

(b) `sum1 = 64`

(c) `sum1 = 120`

(d) `sum1 = 128`

(6). The Command Window output of the following program is

```
clc
clear
while time < 5
      for x = 1:3
            y = time*x^2;
      end
      time = time + 1;
end
time
```

(a) `time = 4`

(b) `time = 5`

(c) `time = 6`

(d) `Undefined function or variable 'time'.`

(7). The Command Window output of the following program is

```
clc
clear
num = 0;
for k = 1:3
      for g = 1:6
            num = num + 1;
      end
end
num
```

(a) `num = 6`

(b) `num = 3`

(c) `num = 18`

(d) `num = 9`

(8). What is the Command Window output of the following code?

```
clc
clear
for i = 1:1:2
     k = 0;
     for j = 1:1:4
          k = k+2;
     end
end
k
```

(a) `k = 16`

(b) `k = 8`

(c) `k = 0`

(d) `k = 2`

(9). The value of `a` to make the output of the last line be `s = 29` is `a =` ________

```
clc
clear
a = ??????;
s = 5;
for i = 1:2
     for k = 1:3
          s = s + a;
     end
end
s
```

(a) `4`

(b) `6`

(c) `24`

(d) `29`

Problem Set

(1). Search any given vector to see if it contains a given number. Display a message in the Command Window each time that value is found and where its location is within the vector. Use the vector `vec = [41 6 2.3 0.2 5 1 5]` and look for the number 6.2 to test the program. Test again with the same vector and the number 5.

(2). Search the given vector, `vec = ['M' 'A' 'T' 'L' 'A' 'B']`, to see if it contains the character `'A'`. Display a message in the Command Window each time that character is found and where its location is within the vector.

Hint: Use the function `strcmp()` to ensure the program will work for other inputs (reference Example 2 in Lesson 5.3).

(3). Write a MATLAB program that generates a vector of 8 elements where each element is a unique random integer in the domain [3, 63].

(4). Generate a matrix of pseudo-random numbers using `rand(n)`, which will create an $n \times n$ matrix. Check each element of the matrix to see whether its value is at least 0.7. If a value does not meet this criterion, immediately skip to the next *row* of the matrix. Output each element that does not meet the criteria to the Command Window in the form: `Value too low for A(i,j) = X`.

(5). Generate a matrix of pseudo-random numbers using `rand(n)`, which will create an $n \times n$ matrix. Check each element of the matrix to see whether its value is at least 0.25. If a value does not meet this criterion, immediately skip to the next *column* of the matrix. Output each element that does not meet the criteria to the Command Window in the form: `Value too low for A(i,j) = X`.

Module 8: Loops

Lesson 8.5 – Working with Matrices and Loops

Learning Objectives

After reading this lesson, you should be able to:

- *reference matrices inside a loop,*
- *store multiple values in a matrix,*
- *access specific portions of a matrix such as the diagonal or the upper triangle,*
- *identify a special matrix.*

This lesson combines several pieces of programming knowledge you have learned and used so far including matrices (Lesson 2.6), conditional statements (Module 5), and nested loops (Lesson 8.4) in MATLAB. In this lesson, you will see some of the nuances and tricks associated with using matrices in loops.

How can I reference matrices in a loop?

As noted in several previous lessons, a vector is a special type of matrix: a one-dimensional matrix. Everything said about matrices in this lesson will also (generally) apply to vectors. Also, recall vectors and matrices as mathematical concepts that were covered in Lesson 4.6 as well as the common MATLAB functions relating to these concepts such as `size()`. We will start by showing examples for referencing both a vector and a matrix in a loop just to be clear.

In this module, we also talked about running a block of code multiple times using loops. In the case of arrays (vectors and matrices), this is useful when referencing many different elements stored inside them: the less “automatic” alternative is referencing every array element manually.

Referencing an array typically means calling out a specific subset of that array. For example, given `vec = [3 7 5.4 8 9]`, we could reference the first three elements with `vec(1:3)`. The values giving the location (e.g., the first and second elements) of the desired elements are often called the index/indices. However, it is common to need to reference each element of a given vector. To do this, we need to use loops.

Example 1

Consider the data given in Table 1 that was collected during axial testing of a metal specimen. In this test, the value of stress and strain is measured at various stages of the experiment. We know we can find Young’s modulus of the metal by finding the relationship between stress and strain.

Table 1: Stress vs. Strain data of a material.

Strain	**Stress (MPa)**
0.001	69
0.002	145
0.003	220
0.004	346
0.005	550

Any element of a vector can be individually “called” for further analysis by using matrix parenthesis notation as discussed above and in Lesson 2.6. Using the same parenthesis notation, a loop can be written to display each stress measurement with the corresponding strain measurement.

Write a program that displays each set of corresponding stress and strain elements in the Command Window.

Solution

MATLAB Code example1.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To display each element of any vector individually
fprintf('To display each element of any vector individually.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
stress = [69.0 145 220 346 550];
strain = [0.001 0.002 0.003 0.004 0.005];

fprintf('   Stress     Strain\n')
disp([stress' strain'])

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
%Note: In this example, i is the index used to reference each element.
fprintf('The individual pairs are:\n')
n = length(stress);

for i = 1:1:n  %Loop through entire vector
    %Displaying each element value
    fprintf('Stress = %3g MPa,  Strain = %4g\n',stress(i),strain(i))
end
```

The two vectors of stress and strain can now be used to generate a stress vs. strain plot, and to find Young's modulus of the metal (see problem 4 of the exercise set of this exercise).

At this point, one might ask, "Why don't we just do `disp(stress)`?" This would not display each element individually. "Well, why not `disp(stress(1))`, `disp(stress(2))`, …" This would not address the "*any*" part of the question. The solution should work for any given vector, which means for any given length (number of elements). So, if you wrote code in that form for a vector with six elements and then tried to use that code on a vector with four elements, MATLAB would return an error. If you used that code on a vector with eight elements, you would get only six out of the eight elements You should try this for yourself using the solution from Example 1.

```
Command Window Output                                    Example 1
PURPOSE
To display each element of any vector individually.

INPUTS
   Stress     Strain
   69.0000    0.0010
  145.0000    0.0020
  220.0000    0.0030
  346.0000    0.0040
  550.0000    0.0050

OUTPUTS
The individual pairs are:
Stress =  69 MPa,  Strain = 0.001
Stress = 145 MPa,  Strain = 0.002
Stress = 220 MPa,  Strain = 0.003
Stress = 346 MPa,  Strain = 0.004
Stress = 550 MPa,  Strain = 0.005
```

Notice that we can reference a vector with a single loop since one of the dimensions will be held at 1. That is, `vec(i,1)` and `vec(i)` are equivalent programming references and evidently only require one iterating (changing) variable in a loop.

In the next example, which uses a matrix, we will need two loops. Perhaps you can guess that it is because both the rows and the columns will need iterators (changing variables). Also, note that we want to "look at" or reference one column or one row of the matrix per loop iteration. Therefore, we will use a nested loop where the parent/outer loop remains constant, while the nested/inner loop runs from the beginning to the- end. In Example 2, this has the effect of looking at an entire row of the matrix before moving on to the next row. Look at the outputs to verify this for yourself!

Example 2

Display each element of any given matrix.

Show a test case for the solution using the matrix $[A] = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

Solution

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To display each element of any given matrix
fprintf('To display each element of any given matrix.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
A = [1 2 3;         %Defining an arbitrary matrix
     4 5 6]

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
rows = size(A,1);  %Finding # of rows of matrix (1st dimension/direction)
cols = size(A,2);  %Finding # of columns of matrix (2nd dimension/direction)

for i = 1:rows       %This loop iterates through each row with index i
    for j = 1:cols   %This loop iterates through each column with index j

        %Displaying each element of the matrix
        fprintf('A(%g,%g) = %g\n',i,j,A(i,j))
    end
end
```

Command Window Output Example 2

```
PURPOSE
To display each element of any given matrix.

INPUTS

A =
     1     2     3
     4     5     6

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

How do I store values in a matrix using a loop?

Often, it is useful to store values obtained in each loop, so you have a vector or matrix of values at the end of the loop(s). While in the previous example we referenced values/elements in an existing matrix, here we create a matrix element-by-element. Figure 1 shows a generalized visualization for storing in a vector, and Example 3 shows a code that stores values in a vector.

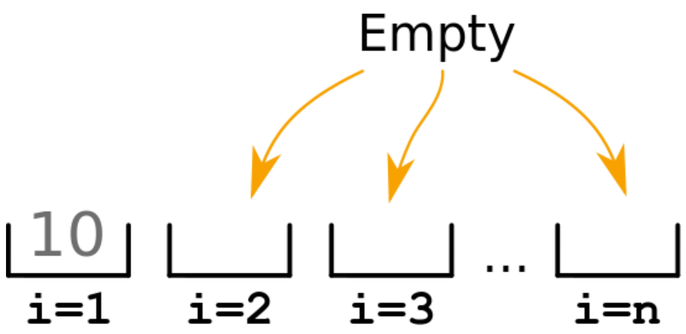


Figure 1: A visualization of the process for generating a vector.

As discussed earlier in this lesson, the outer loop remains constant for one entire completion of the inner loop. We will take advantage of this behavior in Example 3 where we want to average the numbers in each column. Therefore, we want to hold the column number constant (parent loop) while we loop through all the rows of that column with the nested loop. This means that we will go through (average) all the elements in the first column, then the second, and so on.

Example 3

Find the average of each column in any given matrix without using use the built-in functions such as `mean()` and `sum()`. Store the averages in a vector and display the final vector of averages in the Command Window.

Use the matrix given below as a test case.

$$[B] = \begin{bmatrix} 5 & 39 \\ 9 & 14 \\ 3 & 1 \end{bmatrix}$$

Solution

MATLAB Code example3.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find and store the average of each column of a matrix
fprintf('To find and store the average of each column of a matrix.\n\n')
```

MATLAB Code (continued) example3.m

```
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
mat = [5 39;        %Defining an arbitrary matrix (size 3x2)
       9 14;
       3 1];
fprintf('The input matrix is:\n')
disp(mat)

%------------------------------ SOLUTION ----------------------------
[rows, cols] = size(mat);  %Find and store the number of rows and columns

 for j = 1:cols   %Iterating through columns
     colSum = 0;  %Reset each colSum to zero when a new column is started
     for i = 1:rows   %Iterating through rows
         colSum = colSum + mat(i,j); %Adding each value to current colSum
     end
     %avgs is a vector that stores the average of each column
     avgs(j) = colSum/rows;
 end

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
disp('The average of each column is:')
disp(avgs)

disp('Checking the average of each column with the MATLAB function:')
disp(mean(mat,1))  %Check the documentation for syntax on sum()
```

Command Window Output Example 3

```
PURPOSE
To find and store the average of each column of a matrix.

INPUTS
The input matrix is:
     5    39
     9    14
     3     1

OUTPUTS
The average of each column is:
    5.6667   18.0000

Checking the average of each column with the MATLAB function:
    5.6667   18.0000
```

Another real-world example would be if you have a line of code that gets the pressure inside a tank each time the function is run. However, you want to plot and analyze this data changing over time. To do this, you will need to run the sensor read code many times with a loop, and each time the loop runs (the sensor is read), you will need to store that reading in a vector or matrix (depending on how many values are returned by the sensor).

How can I access specific areas of a matrix?

It is useful to understand how to write conditions to reference different parts of a matrix such as upper triangular, lower triangular, and diagonal portions. These three conditions are shown visually in Figure 2. To find these or other locations yourself, just write down the matrix elements, $A(i,j)$, for the area of the matrix you want to reference and look for a pattern. For example, to find the condition for all diagonal elements, just write out $A(1,1)$, $A(2,2)$, $A(3,3)$, $A(4,4)$. You can immediately see diagonal elements always occur when $i = j$.

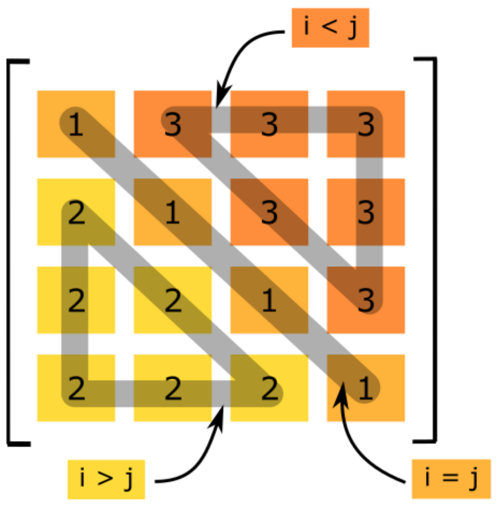


Figure 2: Shows conditions to reference different portions of a matrix.

Examples 4 further demonstrates how to access different locations in a matrix. It also demonstrates how to create a matrix element-by-element, which is just another way of "storing" values. This is a very common task when programming, and therefore, it is essential to fully comprehend. Although the difference of "storing" versus "referencing" may still be foreign, just remember to check which side (left or right) the equals operator (=) the variable in question is on. This will always tell you whether the variable is being referenced or assigned a value (something is being stored in the variable).

Example 4

Write a code that creates a square matrix of the same form like the one shown in Figure 2. The only input is the size of the square matrix.

Solution

MATLAB Code example4.m

```
clc
clear

%------------------------------ PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To create a matrix
fprintf('To create a matrix.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
n = 4;  %Choose size of triangular matrix
fprintf('The size of the matrix should be %gx%g.\n\n',n,n)

%------------------------------ SOLUTION ----------------------------
for i = 1:n                   %Iterate through each row
    for j = 1:n               %Iterate through each column
        %Assign a value to each element (location)
        if i == j             %Location: along the matrix diagonal
            A(i,j) = 1;       %   Setting value of element equal to 1
        elseif i > j          %Location: below diagonal
            A(i,j) = 2;       %   Setting value of element equal to 2
        elseif i < j             %Location: above the matrix diagonal
            A(i,j) = 3;       %   Setting value of element equal to 3
        end
    end
end

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The generated matrix is:\n')
disp(A)                       %Displaying the resulting triangular matrix
```

Command Window Output Example 4

```
PURPOSE
To create a matrix.

INPUTS
The size of the matrix should be 4x4.

OUTPUTS
The generated matrix is:
     1     3     3     3
     2     1     3     3
     2     2     1     3
     2     2     2     1
```

Notice we need two loops to access every element in a matrix: one loop for each matrix direction. For vectors, we needed only one loop to access each element. As you will see in Lesson 8.7 with the sorting example, having a vector as an input does not *always* mean you need only one loop. You need at least one loop to access (or "look at") each element of a vector individually, but depending on the problem, you may need more. A similar statement is true for matrices. One notable, special case is when one only needs to access the diagonal of a matrix. One such application is shown in Example 5.

Example 5

The trace of a matrix is defined only for a square matrix (that is, the number of rows is the same as the number of columns) and is the summation of all its diagonal elements. For a square matrix $[A]$ of size $n \times n$, the trace of a matrix $[A]$ is given by

$$[A] = \sum_{i=1}^{n} a_n$$

Find the trace of any square matrix. The program must also automatically test if the matrix is square.

Test the program with the following two matrices $[A]$ and $[B]$:

$$[A] = \begin{bmatrix} 2 & -1 \\ 1 & 2 \\ 6 & -6 \end{bmatrix} \; [B] = \begin{bmatrix} 5 & 43 & -1 \\ 0 & -13 & 54 \\ 2 & 5 & 29 \end{bmatrix}$$

Solution

MATLAB Code example5.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To find the trace of a matrix
fprintf('To find the trace of a matrix.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
A = [5 43 -1; 0 -13 54; 2 5 29];
fprintf('The input matrix is:\n')
disp(A)

%----SOLUTION----
sqaureFlag = 0;          %Initializing flag value to zero
[rows, cols] = size(A);  %Find and store the number of rows and columns

%If rows do not equal columns, matrix is not square, and the trace
%    cannot be found.
```

MATLAB Code (continued) example5.m

```
if rows ~= cols
    warning('Matrix is not square, so trace of matrix cannot be found.')
else
    sqaureFlag = 1;           %Signal to outputs that the trace was found
    traceSum = 0;             %Intializing trace summing variable
    for i = 1:rows            %Can be rows OR cols since rows==cols
        traceSum = traceSum + A(i,i);  %Summing diagonal values
    end
end

%----------------------------- OUTPUTS -----------------------------
%We can only output the trace if it has been found. A simple solution is to
%   use a flag (although there are other ways).
if sqaureFlag == 1
    fprintf('OUTPUTS\n')
    fprintf('The trace of the matrix is %g.\n',traceSum)
    fprintf('Checking with the MATLAB trace() function, we get %g.\n',...
        trace(A))
end
```

The following Command Window outputs are shown for different matrix inputs. Notice the first Command Window shown has a non-square matrix, which yields an error, while the second Command Window output shown is for a square matrix input.

Command Window Output Example 5

```
PURPOSE
To find the trace of a matrix.

INPUTS
The input matrix is:
     2    -1
     1     2
     6    -6

Warning: Matrix is not square, so trace of matrix cannot be found.
```

Command Window Output Example 5

```
PURPOSE
To find the trace of a matrix.

INPUTS
The input matrix is:
     5    43    -1
     0   -13    54
     2     5    29

OUTPUTS
The trace of the matrix is 21.
Checking with the MATLAB trace() function, we get 21.
```

We can use what we have learned so far in this lesson to test for a specific type of matrix (Example 6). The only extra information one needs to know in these cases is the definition of that special type of matrix, which has little to do with programming skills.

Example 6

Many a time, it is efficient to solve a set of simultaneous linear equations by using iterative schemes such as Gauss-Seidel method. In such cases, the convergence of the solution is guaranteed if the coefficient matrix is strictly diagonally dominant.

To determine if a square matrix is strictly diagonally dominant or not, one compares the row diagonal element to the row non-diagonal elements. For each row, the absolute value of the diagonal element must be *strictly* greater than the sum of the absolute value of the non-diagonal terms. If this condition is true for all rows, only then the matrix is strictly diagonally dominant (SDDM).

Mathematically, a square matrix $[A]$ of $n \times n$ size can be defined as strictly diagonally dominant if

$$|a_{ij}| > \sum_{\substack{j=1 \\ i \neq j}}^{n} |a_{ij}| \quad \text{for } i = 1, 2, ..., n.$$

Take the following matrix as an example

$$[S] = \begin{bmatrix} -5 & 1 & 2 \\ -6 & 12 & 3 \\ 1 & -7 & 9 \end{bmatrix}$$

This is a strictly diagonally dominant matrix because all the rows meet the criteria as shown below.

$$\begin{aligned} |-5| &> |1| + |2| \\ 5 &> 3 \\ |12| &> |-6 + |3| \\ 12 &> 9 \\ |9| &> |1| + |-7| \\ 9 &> 8 \end{aligned}$$

Write a program that determines if a square matrix is strictly diagonally dominant (SDDM) or not.

Test the program using the following two matrices, $[A]$ and $[B]$.

$$[A] = \begin{bmatrix} -5 & 2 & 2 \\ -6 & 12 & 5 \\ 0 & -7 & 9 \end{bmatrix} \quad [B] = \begin{bmatrix} -5 & 3 & 2 \\ -6 & 12 & 6 \\ 0 & -7 & 9 \end{bmatrix}$$

Solution

We need two loops to access each element in the matrix. An `if` statement is used within the nested loop to sum only the non-diagonal elements. Finally, the current row non-diagonal sum is compared to the row diagonal element. If it meets the criteria, it is added to the `validRows` count. This is done in the parent loop (`i`) because the diagonal elements can be referenced with only one loop. Therefore, placing this code inside the nested loop (`j`) would needlessly repeat the code and make the program less efficient.

If at the end of the loops, the number in the `validRows` variable is equal to the number of rows of the matrix, then the condition of the matrix being an SDDM is met.

MATLAB Code example6.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To determine if a matrix is strictly diagonally dominant
fprintf('To determine if a matrix is strictly diagonally dominant.\n\n')

%------------------------------ INPUTS -------------------------------
fprintf('INPUTS\n')
A = [-5 3 2; -6  12  6;0 -7 9];
fprintf('The input matrix is:\n')
disp(A)

%------------------------------ SOLUTION -----------------------------
[rows, cols] = size(A);  %Find and store the number of rows and columns

validRows = 0;           %There are currently 0 validated rows
for i = 1:1:rows
    currentSum = 0;      %Reset sum to 0 for each row
    for j = 1:1:cols
        if i ~= j  %All the non-diagonal elements
            %Summing non-diagonal elements
            currentSum = currentSum + abs(A(i,j));
        end
    end

    %Comparing the diagonal element to non-diagonal summation
    if abs(A(i,i)) > currentSum
        validRows = validRows + 1;  %Count the number of valid rows
    end
end
```

MATLAB Code (continued) example6.m

```
%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
if validRows == rows  %All the rows must meet SDDM conditions
    fprintf('Matrix is strictly diagonally dominant.\n')
else
    fprintf('Matrix is NOT strictly diagonally dominant.\n')
end
```

The following Command Window outputs are shown for different matrix inputs. As you can see, the first matrix input is strictly diagonally dominant, while the second is not.

Command Window Output Example 6

```
PURPOSE
To determine if a matrix is strictly diagonally dominant.

INPUTS
The input matrix is:
    -5     2     2
    -6    12     5
     0    -7     9

OUTPUTS
Matrix is strictly diagonally dominant.
```

Command Window Output Example 6

```
PURPOSE
To determine if a matrix is strictly diagonally dominant.

INPUTS
The input matrix is:
    -5     3     2
    -6    12     6
     0    -7     9

OUTPUTS
Matrix is NOT strictly diagonally dominant.
```

What is vectorization?

Although it can be a good exercise to perform matrix operations like addition, subtraction, and multiplication the "long way" (i.e., the non-vectorized way) when first learning to use loops and work with matrices, it is not standard practice.

Vectorization is the practice of implementing elemental operations as matrix operations in programs. For example, if you have a system of three equations and three unknowns, you would vectorize the solution by putting the equations into matrix form, and find the solution using linear algebra. For many mathematical matrix operations, you should implement them in MATLAB using "vectorization" rather than loops as this takes advantage of the efficiency of MATLAB, and hence avoid unnecessary complexity in the code.

There are many examples where you can vectorize operations involving matrices in MATLAB. They include matrix addition, subtraction, and multiplication. More complex examples are when you combine these operations into equations. This occurs often in programming in areas like machine learning and state-space controls.

How can I vectorize matrix operations in MATLAB

The words "vectorize" and vectorization" may be new to you, but the concepts and syntax are not. Recall that Lesson 4.6 on Linear Algebra was where we first learned how to evaluate mathematical expressions and equations that contain matrices. We did not know anything about loops then, and therefore, the code we wrote was a vectorized solution! We did this for simple things like addition, subtraction, and multiplication of matrices as well as more complex tasks like solving systems of equations in matrix form. We recommend reviewing the examples from Lesson 4.6 to more fully grasp vectorization.

Examples 7 and 8 compare vectorized and non-vectorized solutions for matrix operations, which of course will return the same answers. Although it can be a good exercise to perform matrix operations like addition, subtraction, and multiplication the "long way" (i.e., the non-vectorized way) when first learning to use loops and work with matrices, it is not standard practice. When writing professional code, one should use vectorized solutions in MATLAB whenever possible. This is because it is the vectorized syntax that has been optimized to be faster than the equivalent element-by-element loop operations we could write.

Example 7

Write a program that adds any two matrices (given that the matrices are of equal size) using element-by-element loops operations and vectorized operations. Use the matrices $[A]$ and $[B]$ to test your program.

$$[A] = \begin{bmatrix} 1 & 5 & 8 \\ 12 & 6 & 3 \\ 19 & 45 & 0 \end{bmatrix} \quad [B] = \begin{bmatrix} 14 & 75 & 4 \\ 11 & 13 & 23 \\ 0.5 & 8 & 3 \end{bmatrix}$$

MATLAB Code example7.m

```
clc
clear

%---------------------------- PURPOSE ----------------------------
fprintf('PURPOSE\n')
%To perform matrix addition
fprintf('To perform matrix addition.\n\n')
```

MATLAB Code (continued) example7.m

```
%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
fprintf('The two input matrices are:\n')
%Defining two matrices
A = [1 5 8; 12 6 3; 19 45 0];
B = [14 75 4; 11 13 23; 0.5 8 3];
fprintf('A = \n')
disp(A)
fprintf('B = \n')
disp(B)

%------------------------ SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
%***Non-vectorized solution***
if size(A) == size(B)  %Check to see if addition is possible
    for row = 1:size(A,1)
        for col = 1:size(A,2)
            C(row,col) = A(row,col) + B(row,col);
        end
    end
end
fprintf('The non-vectorized solution is:\n')
disp(C)  %Display C after loops have finished to avoid unnecessary
         %   Command Window outputs

%***Vectorized solution***
D = A + B;
fprintf('The vectorized solution is:\n')
disp(D)
```

Command Window Output Example 7

```
PURPOSE
To perform matrix addition.

INPUTS
The two input matrices are:
A = 
     1     5     8
    12     6     3
    19    45     0

B = 
   14.0000   75.0000    4.0000
   11.0000   13.0000   23.0000
    0.5000    8.0000    3.0000
```

Command Window Output (continued) Example 7

```
OUTPUTS
The non-vectorized solution is:
   15.0000   80.0000   12.0000
   23.0000   19.0000   26.0000
   19.5000   53.0000    3.0000

The vectorized solution is:
   15.0000   80.0000   12.0000
   23.0000   19.0000   26.0000
   19.5000   53.0000    3.0000
```

You should now be able to see why, in their documentation on vectorization, MATLAB lists the compactness of a vectorized matrix operation as improving the appearance of the code and reducing the complexity (and hence the likelihood of errors).

Example 8

Write a program that finds the element-by-element product of any two square matrices of equal size using loops operations and vectorized operations. Use the matrices $[A]$ and $[B]$ to test your program.

$$[A] = \begin{bmatrix} 1 & 5 & 8 \\ 12 & 6 & 3 \\ 19 & 45 & 0 \end{bmatrix} \quad [B] = \begin{bmatrix} 14 & 75 & 4 \\ 11 & 13 & 23 \\ 0.5 & 8 & 3 \end{bmatrix}$$

Solution

MATLAB Code example8.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To multiply two matrices element-by-element
fprintf('To multiply two matrices element-by-element.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
fprintf('The two input matrices are:\n')
%Defining two matrices
A = [1 5 8; 12 6 3; 19 45 0];
B = [14 75 4; 11 13 23; 0.5 8 3];
fprintf('A = \n')
disp(A)
fprintf('B = \n')
disp(B)
```

MATLAB Code (continued) example8.m

```
%------------------------- SOLUTION/OUTPUTS -------------------------
fprintf('OUTPUTS\n')
%***Non-vectorized solution***
if size(A) == size(B)  %Check to see if element-by-element matrix
                       %    multiplication is possible
    for row = 1:size(A,1)
        for col = 1:size(A,2)
            C(row,col) = A(row,col)*B(row,col);
        end
    end
end

fprintf('The non-vectorized solution is:\n')
disp(C)  %Display C after loops have finished to avoid unnecessary
         %   Command Window outputs

%***Vectorized solution***
D = A.*B;
fprintf('The vectorized solution is:\n')
disp(D)
```

Command Window Output Example 8

```
PURPOSE
To multiply two matrices element-by-element.

INPUTS
The two input matrices are:
A =
     1     5     8
    12     6     3
    19    45     0

B =
   14.0000   75.0000    4.0000
   11.0000   13.0000   23.0000
    0.5000    8.0000    3.0000

OUTPUTS
The non-vectorized solution is:
   14.0000  375.0000   32.0000
  132.0000   78.0000   69.0000
    9.5000  360.0000         0

The vectorized solution is:
   14.0000  375.0000   32.0000
  132.0000   78.0000   69.0000
    9.5000  360.0000         0
```

What are some tips I can use for vectorization?

Below are a few tips to keep in mind when you are trying to vectorize equations containing matrices/vectors.

- Use short variable names for vectors and matrices when possible.

- Be sure to know what you *want* to happen and what *is* happening during the evaluation (calculation) of your equations in MATLAB. This always applies to programming.
- Make sure you understand and appropriately use array and matrix operations (common mistake).
- Use the shorthand (') for the transpose of a matrix.

Multiple Choice Quiz

(1). The Command Window output of the following program is

```
clc
clear
y = [1 14 5 8 9];
for i = 1:1:length(y)-1
      s(i) = y(i) + y(i+1);
end
s(4)
```

(a) `ans = 17`

(b) `ans = 15`

(c) `ans = 13`

(d) `ans = 19`

(2). The Command Window output of the following program is

```
clc
clear
r = [1 2 8 5];
u = [3 2 6 3];
for i = 1:1:4
      if r(i) > u(i)
            q(i) = r(i) + u(i);
      else
            q(i) = r(i);
      end
end
q
```

(a) `q = [1 4 14 8]`

(b) `q = [1 2 8 5]`

(c) `q = [1 2 14 8]`

(d) `q = [4 2 6 3]`

(3). The Command Window output of the following program is

```
clc
clear
r=[1 2 8 5];
u=[3 2 6 3];
for i = 1:1:4
    if r(i) > u(i)
        break
    else
        q(i) = r(i);
    end
end
q
```

(a) `q = [1 2]`

(b) `q = [3 2]`

(c) `q = [1 2 8 5]`

(d) `q = [3 2 6 3]`

(4). The Command Window output of the following program is

```
clc
clear
a = [1 0 0; 0 1 0; 0 0 1];
count = 0;
[rows,cols] = size(a);
for i = 1:1:rows
    for j = 1:1:cols
        if i == j && a(i,j) == 1
            count=count+1;
        end
        if i ~= j && a(i,j) == 0
            count=count+1;
        end
    end
end
count
```

(a) `count = 0`

(b) `count = 3`

(c) `count = 6`

(d) `count = 9`

(5). The Command Window output of the following program is

```
clc
clear
a = [3 3 0; 0 1 2; 0 0 -1];
sum1 = 0;
[rows,cols] = size(a);
for i = 1:1:rows
    for j = 1:1:cols
        if i ~= j
             sum1 = sum1 + a(i,j);
        end
    end
end
sum1
```

(a) `sum1 = 2`

(b) `sum1 = 3`

(c) `sum1 = 5`

(d) `Undefined function or variable...`

(6). The Command Window output of the following program is

```
clc
clear
a = [2 7 2; 8 -1 2; 0 10 1; 2 -4 7];
[rows,cols] = size(a);
negNum = 0;
for i = 1:1:rows
    for j = 1:1:cols
        if i == j && a(i,j) < 0
             negNum = a(i,j);
             break
        end
    end
end
negNum
```

(a) `negNum = 0`

(b) `negNum = -4`

(c) `negNum = -1`

(d) `Undefined function or variable...`

Problem Set

(1). Using nested loops, write a program that outputs the sum, $[A] + [B]$ of two equally sized matrices, $[A]$ and $[B]$. Test and run your program for the following two matrices, $[A]$ and $[B]$.

$$[A] = \begin{bmatrix} 0 & 5 & -3 \\ 1 & -7 & 9 \\ 5 & 5 & 12 \end{bmatrix} \quad [B] = \begin{bmatrix} 14 & 2 & 5 \\ 1 & 7 & 3 \\ 1 & 3 & 5 \end{bmatrix}$$

(2). Using nested loops, write a function that outputs the numeric value of the difference of two equally sized matrices, $[A]$ and $[B]$. Test and run your program for the two following matrices, $[A]$ and $[B]$.

$$[A] = \begin{bmatrix} 12 & 13 & 10 \\ 7 & 9 & 13 \\ 1 & 8 & 12 \end{bmatrix} \quad [B] = \begin{bmatrix} -14 & 10 & 5 \\ -7 & 7 & 4 \\ 2 & 0 & -9 \end{bmatrix}$$

(3). Write your own function, `myDot`, that finds the vector dot product of two input vectors. The function inputs are two vectors, `vec1` and `vec2`, and the output is the value of the vector dot product. Do not use the `dot()` or similar MATLAB functions.

Test your function for two vectors

```
vec1 = [3  -7  23]
vec2 = [-7  0  12]
```

(4). Without using the `sum()` function, write a function (`myYoung`) that outputs Young's modulus of a material based on an input of stress vs. strain data. The stress and strain values must be entered using two vectors, `stress`, and `strain`. The Young's modulus (E), given the stress (σ) and strain (ε) values, is found as

$$E = \frac{\sum_{i=1}^{n} \sigma_i \varepsilon_i}{\sum_{i=1}^{n} (\varepsilon_i)^2}$$

Test your function with the data provided in Table A.

Table A: Strain and Stress data of a material.

Strain (m/m)	**Stress (MPa)**
0.00010	19.10

Strain (m/m)	**Stress (MPa)**
0.00012	22.81
0.00100	187.0
0.00150	284.2
0.00180	344.3
0.00220	417.0
0.00260	495.0

(5). Write a program that determines if a square matrix, A is symmetric or not. A symmetric matrix is where $a_{ij} = a_{ji}$ for all i, j. Matrix $[A]$, given below, is an example of a symmetric matrix, which you can use to test your solution. The program needs to work for a square matrix of any size.

$$[A] = \begin{bmatrix} 2 & 8 & 9 \\ 8 & 4.5 & 6 \\ 9 & 6 & -1 \end{bmatrix}$$

(6). Write a program that outputs the value of the summation of the perimeter elements (outer elements) of a rectangular matrix. Use your knowledge of loops and/or conditional statements to write the program.

The program input is:
1) a rectangular matrix, `rectMat`.

The program output is:
1) the value of the summation of the perimeter elements, `perimeterSum`.

Test and run your program for the following 3×4 matrix, `rectMat`.

$$\begin{bmatrix} 2 & -3 & 1 & 0 \\ 5 & 7 & 1 & -4 \\ 9 & -6 & 0 & 2 \end{bmatrix}$$

Hint: The summation of the perimeter values for the above matrix is,

$2 + (-3) + 1 + 0 + (-4) + 2 + 0 + (-6) + 9 + 5 = 6.$

(7). The column sum norm of a rectangular matrix $[A]$ with m rows and n columns is defined as

$$norm[A] = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}|$$

In other words, find the sum of the absolute value of the m elements of each of the n columns. Then find the maximum of these n values (do not use `max()`, `min()` or similar MATLAB function). This maximum value is the norm of the matrix [A]. Using your knowledge of loops and/or conditional statements, write the program that finds the norm of an input matrix. Do not use `norm()` and other similar MATLAB functions.

The program input is:
1) a rectangular matrix, `mat`.

The program output is:
1) the norm of the matrix, `normMat`.

Test and run your program with the following matrix, `mat`.

$$\begin{bmatrix} 25 & 20 & 3 & 2 \\ 5 & -10 & 15 & -25 \\ 6 & 16 & -7 & 27 \end{bmatrix}$$

Hint: Consider the following 2×3 matrix, A

$$[A] = \begin{bmatrix} 6 & 3 & -4 \\ 2 & 7 & 1 \end{bmatrix}.$$

the *norm[A]* is found as follows:
Add absolute values of elements of first column, that is, $|6| + |2| = 8$.
Add absolute values of elements of second column, that is, $|3| + |7| = 10$.
Add absolute values of elements of third column, that is, $|-4| + |1| = 5$.
The maximum then of the first, second and third column sum values is maximum of $(8,\ 10,\ 5) = 10$. The norm of matrix $[A]$, hence, is 10.

(8). A square matrix is considered bisymmetric if it is symmetric about *both* of its main diagonals. For example, consider the following 5×5matrix:

$$\begin{bmatrix} D_1 & b & c & d & A_1 \\ b & D_2 & e & A_2 & d \\ c & e & D_3 & e & c \\ d & A_2 & e & D_2 & b \\ A_1 & d & c & b & D_1 \end{bmatrix}$$

This matrix is symmetric about both main diagonals D and $A,$ and therefore considered bisymmetric. Use your knowledge of programming concepts (loops and conditional statements) to write a program that outputs whether a square matrix is bisymmetric or not.

The program input is:
1) a square matrix, `mat`.

The program output is:
1) "Bisymmetric matrix" or "Not a bisymmetric matrix"

Test and run your program for the two matrices given on the next page.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 6 & 7 & 8 & 4 \\ 3 & 7 & 9 & 7 & 3 \\ 4 & 8 & 7 & 6 & 2 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix} \; and \; \begin{bmatrix} 3 & 3 & 2 & -1 \\ 3 & 4 & 6 & 0 \\ 2 & 6 & 4 & 3 \\ -1 & 0 & 3 & 7 \end{bmatrix}$$

(9). Using loops, write a program that transposes an input row vector. Do not use the `transpose()` or equivalent function. The transpose of a row vector is a column vector. For example

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

.

Test your program with the following vector.
```
a = [6  -6  4  13  0  7  2]
```

(10). Two matrices $[A]$ and $[B]$ can be multiplied only if the number of columns of $[A]$ is equal to the number of rows of $[B]$. If [A] is a $m \times p$ matrix and $[B]$ is a $p \times n$ matrix, then the resulting matrix $[C]$ is a $m \times n$ matrix. So how does one calculate the elements of $[C]$ matrix?

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots\ldots + a_{ip} b_{pj}$$

for each $i = 1,\ 2, \ldots\ldots,\ m, \text{and} j = 1,\ 2, \ldots\ldots,\ n.$

To put it in simpler terms, the i^{th} row and j^{th} column element of the $[C]$ matrix in $[C] = [A][B]$ is calculated as multiplying the i^{th} row of $[A]$ by the j^{th} column of $[B]$, that is,

$$\begin{aligned} c_{ij} &= \begin{bmatrix} a_{i1} & a_{i2} & \cdots & \cdots & a_{ip} \end{bmatrix} \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ \vdots \\ b_{pj} \end{bmatrix} \\ &= a_{i1}\ b_{1j}\ +\ \ a_{i2}\ b_{2j} +\ \cdots\cdots +\ a_{ip}\ b_{pj}. \\ &= \sum_{k=1}^{p} a_{ik} b_{kj} \end{aligned}$$

Complete the following:

(a). Write a function, `myMult`, that outputs the result of the multiplication of two matrices. The inputs are

1. first matrix, `[A]`,
2. second matrix, `[B]`.

The output is

1. the matrix `[C]`, where `[C] = [A][B]`

If the matrices cannot be multiplied, the output needs to be:
`The matrices cannot be multiplied.`

(b). Conduct the following tests of the function `myMult` in a testing m-file

1. Given

$$[A] = \begin{bmatrix} 5 & 2 & 3 \\ 1 & 2 & 7 \end{bmatrix},$$

$$[B] = \begin{bmatrix} 3 & -2 \\ 5 & -8 \\ 9 & -10 \end{bmatrix},$$

find

$$[C] = [A][B]$$

2. Conduct two more appropriate tests of the function.

Module 8: Loops

Lesson 8.6 – Applied Loops

Learning Objectives

After reading this lesson, you should be able to:

- *sort matrices and vectors,*
- *search matrices and vectors,*
- *sum matrices and vectors,*
- *plot inside of a loop (repeated plotting),*
- *use common programming tricks for loops.*

Why is this lesson important?

There are a few common tasks using loops that are essential to understand how to implement them manually. We will cover how to sort, sum, and search in a vector (and a few other things). These have corresponding MATLAB functions that do not require you to do any of this, but knowing how to do these things not only gives you more practice with the programming concepts we have learned, but it also enables you to implement parts of these strategies in other solutions you write (that are not available as predefined functions).

Also, note that the content of this lesson is mostly contained in the example programs, so it is imperative to have a good understanding of these.

How can I sort an array?

Although there are predefined functions in MATLAB like `sort()`, it is useful to understand how to sort a vector manually. Bubble sort compares two adjacent elements and determines through a condition whether they need to be swapped to have a sorted vector.

Example 1

Sort any given vector of numbers from smallest to largest. Test the program with the vector

`[v] = [10 4 99 -7].`

Note you do not need to use the iterator variable (`repeat` in this case) for the parent loop to run.

Solution

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To sort a vector
fprintf('To sort a vector.\n\n')

%------------------------------ INPUTS ------------------------------
fprintf('INPUTS\n')
vec = [10 4 99 -7];  %Defining vector to sort
fprintf('The unsorted vector is:\n')
disp(vec)

%------------------------------ SOLUTION ----------------------------
n = length(vec);   %Getting number of elements in vector

%This loop repeats the whole sorting process
for repeat = 1:n-1  %It is only possible to do n-1 comparisons on n data
                    %    points
    %This loop scans through the vector once
    for i = 1:n-1
        %Checks to see if elements need to be switched
        if vec(i) > vec(i+1)  %Changes depending on ascending or descending
                              %    sort
            temp = vec(i);    %Need to save the value stored at vec(i)
                              %   before it is overwritten in the next line
            vec(i) = vec(i+1);
            vec(i+1) = temp;
        end
    end
end

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The sorted vector is:\n')
disp(vec)
```

```
Command Window Output                                          Example 1
PURPOSE
To sort a vector.

INPUTS
The unsorted vector is:
    10     4    99    -7

OUTPUTS
The sorted vector is:
    -7     4    10    99
```

The nested loop (`i`) in Example 1 could be made more efficient by referencing the `repeat` iterator: `for i = repeat:n-1`. This is because on the first iteration of `repeat, i = 1:n-1`, and the first element of `vec` is guaranteed to be correctly sorted at the end of that iteration. On the second iteration of `repeat, i = 2:n-1`, and the first and second elements of `vec` are guaranteed to be correctly sorted at the end of that iteration. You can see that there is a pattern here that we could take advantage of to write the code more efficiently.

In Figure 1, you can see a visual representation of how two elements are switched in the bubble sort method seen in Example 1.

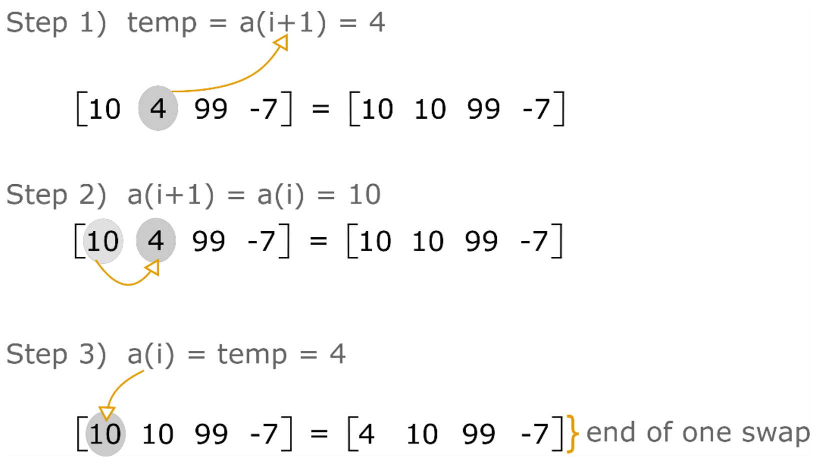


Figure 1: Visualization of how two elements are swapped in the bubble sort method.

In Example 2, we will apply the ideas covered in Example 1 to a matrix. As noted in the example, this is as simple as adding another for loop to repeat the process for each “vector” (row or column depending on your choice) in the matrix.

Example 2

Sort the columns of any given matrix. Test the solution using matrix $[B]$.

$$[B] = \begin{bmatrix} 17 & 5 & 4 \\ 9 & 23 & 0 \\ -2 & 6 & 1.1 \end{bmatrix}$$

Treat each column as a vector. All we need to do is repeat the vector sorting process for each column in the matrix. Therefore, we only need to nest Example 1 inside another loop.

Solution

MATLAB Code example2.m

```matlab
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To sort the columns of a matrix
fprintf('To sort the columns of a matrix.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
B = [17 5 4;
     9 23 0;
     -2 6 1.1]; %Defining vector to sort
fprintf('The unsorted matrix is:\n')
disp(B)

%----------------------------- SOLUTION ----------------------------
[rows, cols] = size(B); %Finding and storing the number of rows and columns

for j = 1:cols
    B(:,j); %Column to be sorted (unsuppress to see)

    %%START vector sorting process (Example 1)
    for repeat = 1:rows-1
        %This loop scans through the column once
        for i = 1:rows-1
            %Remember j will remain constant while this loop runs
            if B(i,j)>B(i+1,j)
                temp = B(i,j);
                B(i,j) = B(i+1,j);
                B(i+1,j) = temp;
            end
        end
    end
    %%END vector sorting
end

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The sorted matrix is:\n')
disp(B)
```

```
Command Window Output                                          Example 2
PURPOSE
To sort the columns of a matrix.

INPUTS
The unsorted matrix is:
   17.0000    5.0000    4.0000
    9.0000   23.0000         0
   -2.0000    6.0000    1.1000
```

```
Command Window Output (continued)                              Example 2
OUTPUTS
The sorted matrix is:
   -2.0000    5.0000         0
    9.0000    6.0000    1.1000
   17.0000   23.0000    4.0000
```

How can I find the sum of a vector?

It is relatively simple to find the sum of a vector manually, and there is even a built-in MATLAB function for it called `sum()`. In fact, we have already covered summing in a loop when we covered `while` loops (Lesson 8.1). However, it is important to pay close attention to the next example as it takes advantage of a key property of loops to create a summing mechanism, which is itself extremely common.

Example 3

Find the sum of any given vector. Do not use the built-in MATLAB functions such as `sum()`.

Test the program with the vector: $[a] = [1 \;\; 565 \;\; 4 \;\; 5 \;\; 8 \;\; 9 \;\; 22]$.

Solution

MATLAB Code example3.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To sum all the elements of a vector
fprintf('To sum all the elements of a vector.\n\n')

%----------------------------- INPUTS ------------------------------
fprintf('INPUTS\n')
vec    = [1 565 4 5 8 9 22]; %Defining an arbitrary vector
fprintf('The input vector is:\n')
disp(vec)

%----------------------------- SOLUTION ----------------------------
sumVar = 0; %Defining summing variable

%Looping through all the elements in the vector
for i = 1:length(vec)
    sumVar = sumVar + vec(i); %Add each element to the summing variable
end
```

MATLAB Code (continued) example3.m

```
%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
%Using a MATLAB's built-in sum() function to check if our answer is correct.
correctSum = sum(vec);

if sumVar == correctSum   %Checking to see if our answer is the same as
                          %    correct answer
    disp('We are correct!')
    fprintf('The sum of the vector is %g.\n',sumVar)
else                      %If it is not the same, it must be wrong.
    disp('Something went wrong! The sum is incorrect.')
end
```

Command Window Output Example 3

```
PURPOSE
To sum all the elements of a vector.

INPUTS
The input vector is:
     1   565     4     5     8     9    22

OUTPUTS
We are correct!
The sum of the vector is 614.
```

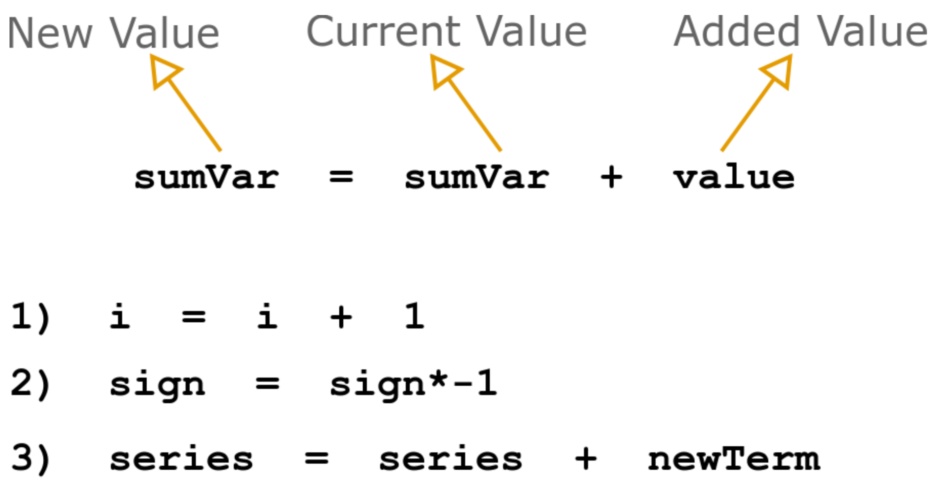


Figure 2: General form of changing the current value of a value and update the variable to the new value. Below that, you can see three applications of the concept.

Some use cases for calling a variable and assigning it a new value on the same line (see also Figure 2) are:

- `i = i + 1`
- `sign = sign*(-1)`
- `series = series + newTerm`

One can see that the same fundamental concept of calling and assigning variables is used in each case. When a variable appears on the right side of the equals sign, the variable is being called (we are asking for its current value). When a variable appears on the left side of the equals sign, we are assigning the variable a new value. One needs to fully grasp this concept.

You can get more practice with summing things in a loop by using different types of series. Also, remember summing can be implemented in a matrix by recognizing that matrices are just collections of vectors (columns/rows depending on how you would like to sort).

How can I plot different variations of a function using loops?

In some cases, we need to plot many different sets of data points, and to avoid long chains of `plot()` calls, we can use loops to run one `plot()` call multiple times. (If you need a quick review on plotting, jump to Lesson 3.1.) A few examples of this are changing part of an equation, hence, changing the vector of data and the equation to the plot. Example 6 shows the first of these cases where we multiply a function by a different constant in each iteration and show them all on the same plot (Figure 3).

✓ ***Important Note:*** In general, you should avoid placing lines of code inside loops that never change between the beginning and end of the loop as this would make your code less efficient.

Example 4

Using only one `plot()` call and a for loop, plot the mathematical function, $y(x) = ke^x$ from $x = 0$ to 6 for $k = 1, 2, ..., 5$.

Solution

MATLAB Code example4.m

```
clc
clear
close all

%----------------------------- PURPOSE -----------------------------
%To plot multiple forms of a function

%----------------------------- INPUTS ------------------------------
x = 0:0.2:6;        %Range to plot the function over

%------------------------ SOLUTION/OUTPUTS -------------------------
figure
hold on  %Declaring "hold on" so all plots will appear on the same figure
for coef = 1:4
    fx = coef*exp(x);   %Function we want to plot has coefficient change
                        %    each loop iteration
    plot(x,fx)          %Plot each new function
    legendLabels{coef} = [num2str(coef) '*e^x'];  %Store labels to use
                                                  %    in legend()
    %Note: we need to use a cell array here because we want to store
    %    strings as elements.
end
hold off
```

MATLAB Code (continued) example4.m

```
%Figure Formatting
legend(legendLabels,'Location','NorthWest')       %Use stored labels
ylabel('\itf(x)')
xlabel('\itx')
```

As shown in Figure 3, all five functions are plotted on one plot. One can see that as the value of k increases, the slope of the function increases. To get appropriate legend labels in an automated way, we make use of cell arrays, which are, in structure, like the vectors and matrices we are familiar with except cell arrays can hold non-numeric values like strings of text.

To create our legend on the plot, we simply assign `legend()` our cell array of strings. If you do not fully grasp the concept of a cell array, that is ok because it is not the focus of this lesson.

!](./Chapter0806Lesson100AppliedLoopsMedia/media/image5.png){width="3.7604166666666665in" height="2.967199256342957in"}
Figure 3: Figure output for the code given in Example 4.

Although this lesson is certainly not an exhaustive list of common techniques you might use in loops, it should give you an idea of what to be on the lookout for. This module should also serve as a lesson in how important experiential programming skills are.

In the first half of the module, we introduced new syntax related to the fundamental programming concept of loops (iterating code). In the second half of this module, we have covered many different programming solutions using loops that covered important skills and knowledge but without introducing much new syntax. This should encourage and motivate you to commit these and any other programming techniques to memory.

Multiple Choice Quiz

(1). The Command Window output of the following program is

```
clc
clear
sum1 = 0;
for i = 1:1:3
    for j = 1:1:3
        sum1 = sum1 + 4;
    end
end
sum1
```

(a) `sum1 = 0`

(b) `sum1 = 4`

(c) `sum1 = 12`

(d) `sum1 = 36`

(2). The Command Window output of the following program is

```
clc
clear
sum1 = 1;
for i = 1:1:2
    for j = 1:1:3
        sum1 = sum1*2;
    end
end
sum1
```

(a) `sum1 = 18`

(b) `sum1 = 64`

(c) `sum1 = 120`

(d) `sum1 = 128`

(3). The Command Window output of the following program is

```
clc
clear
A = [5 4 1 2];
for i = 1:1:3
    if A(i) > A(i+1)
        A(i+1) = A(i);
        A(i) = A(i+1);
    end
end
A
```

(a) `A = [4 1 2 5]`

(b) `A = [5 4 1 2]`

(c) `A = [5 5 5 5]`

(d) `A = [1 2 4 5]`

(4). The Command Window output of the following program is

```
clc
clear
A = [5 4 1 2];
for i = 1:1:3
    if A(i) > A(i+1)
        temp = A(i+1);
        A(i+1) = A(i);
        A(i) = temp;
    end
end
A
```

(a) `A = [4 1 2 5]`

(b) `A = [5 4 1 2]`

(c) `A = [5 5 5 5]`

(d) `A = [1 2 4 5]`

(5). The Command Window output of the following program is

```
clc
clear
vec = [5 10 15 20];
vecSum = 0;
for i = 1:2
      for j = 1:length(vec)
            vecSum = vecSum + vec(j);
      end
end
vecSum
```

(a) `vecSum = 50`

(b) `vecSum = 150`

(c) `vecSum = 100`

(d) `Undefined function or variable 'vecSum'.`

(6). The Command Window output of the following program is

```
clc
clear
A = [5 4 1 2];
for j = 1:1:4
    for i = 1:1:3
        if A(i) > A(i+1)
            temp = A(i);
            A(i) = A(i+1);
            A(i+1) = temp;
        end
    end
end
A
```

(a) `A = [4 1 2 5]`

(b) `A = [5 4 1 2]`

(c) `A = [5 5 5 5]`

(d) `A = [1 2 4 5]`

Problem Set

(1). Using your knowledge of loops and/or conditional statements, write a MATLAB program that determines the value of the following infinite series

$f(x) = x + \frac{1}{2}x^2 + \frac{1}{3}x + \frac{1}{4}x^2 + ...$

There are two program inputs

1. the value of x, and
2. the number of terms to use.

There is one program output

1. the numeric value of the series.

Your program must work for any set of inputs. Assume that the value for the number of terms to use will always be entered as a positive whole number.

Test your program for the following set of inputs:
number of terms = 26
value of x = 0.67

(2). Write a program to numerically sort any input vector, `vec`, from smallest to largest. You may use the bubble sort program shown in Example 1 of this lesson as a guide. Test and run your program for the following vector `vec = [8, 2, 16, 12, 2, 5]`.

(3). Write a program to numerically sort any input vector, `vec`, from largest to smallest. You may modify the bubble sort program shown in Example 1 of this lesson. Test and run your program for the following vector `vec = [8, 2, 16, 12, 2, 5]`.

(4). A typical alarm clock has the following features: (1) alarm set time, (2) current time, and (3) desired "snooze" time. Use your knowledge of loops (for-end and/or while-end) and conditional statements to write a MATLAB program that outputs the wake up time of an individual using an alarm clock. The program inputs are:

1. Alarm set time (*disregard* AM and PM setting), `alarmSet` as a vector with 1 row and 2 columns, with the first element being the hour and the second element being the minutes.
2. Initial "snooze" time, `snooze`. (This time value is always ≥ 4 minutes)
3. Number of times the "snooze" button is used, n. ($0 \leq n \leq 50$)

Each time the snooze button is used after the first use, the snooze time decreases by 1 minute. However, the **minimum** snooze time is 4 minutes.

The program output is (after the snooze button is hit n times):

1. Final wake up time, wakeup as a vector with 1 row and 2 columns, with the first element being the hour and the second element being the minutes.

Test the program using this example: If the alarm set time is at 7:30 and is entered as `alarmSet = [7 30]`, the initial snooze time is set at 8 minutes, and the snooze button was hit 6 times, the wake up time is 8:04 (overall snooze time is: 8+7+6+5+4+4=34), hence `wakeup = [8 4]`.

(5). Using your knowledge of loops and/or conditional statements, write a MATLAB program that conducts the following summation,

$$\sigma = \sum_{j=1}^{n} \left(ax(j) + e^{y(j)} \right)$$

where,

$$a = \text{real constant}$$

$$y = 0.367aj$$

$$x = \begin{cases} 4.5y + 0.5jy < 2 \\ jy \geq 2 \end{cases}$$

The program has two inputs, the number of terms to use, n and the value of the constant, a. The program has one output, the numeric value of the summation. Do not use the `sum` or similar MATLAB function to complete this problem.

Test and run your program for the following set of inputs: `a = 0.75` and `n = 30`.

(6). Without using the `max()`, `min()`, or `sort()` functions, write your own MATLAB program that finds the minimum and maximum element of any input vector. Output the original vector, and the minimum and maximum elements. Test your program for the following input vector `vec = [12, 9, -10, 8, 8.9, -7.8, 15]`.

(7). The secant method is used to approximate the value of the root(s) of an equation $f(x)$=0. The secant method requires the user to make two initial guesses x_0 and x_1 of the root of the equation, but which do not necessarily need to bracket the root. The secant method iterative formula is given by

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k), \quad k = 0, 1, 2, 3, ...$$

where,
x_{k+1} is the current approximation,
x_k is one of the previous approximations,
x_{k-1} is the other previous approximation.

This process is repeated until the root is found.

Write a program that uses the secant method to find the approximate root(s) of an equation. The program inputs are, the function f, the first two guesses, `x(1)` and `x(2)`, and the number of iterations to conduct, n. The output is the approximate value of the root of the equation, `rootVal`.

Although not required, use vectors to store the approximations of the root. Your developed vector will contain `n+2` elements.

Hint: Be sure not to use same numbers for the initial two guesses, and only use a reasonable number of loop repetitions.

(8). Without using the `max()`, `min()`, or similar MATLAB functions, write a program that outputs the maximum and minimum element values of any rectangular matrix.

The program input is:

1. a rectangular matrix, `mat`.

The program outputs are:

1. the maximum value, `big`, and
2. the minimum value, `small`.

Test and run your program for the following matrix, `mat`.

$$\begin{bmatrix} -1 & 0 & 2 \\ 7 & -4 & 12 \\ 4 & 8.2 & 11 \\ -11 & 0 & -12 \end{bmatrix}$$

(9). Write a program in MATLAB that for a square matrix subtracts the *sum* of the absolute value of the elements *above* the diagonal from the *product* of the absolute value of the diagonal elements. Do not use the sum() or any other similar MATLAB functions to solve this problem. Use your knowledge of loops and/or conditional statements to write the program.

The program input is:

1. a square matrix, `A`.

The program output is:

1. the value of the difference, `processedMat`.

Test and run your program for the following 4×4 matrix, `A`.

$$\begin{bmatrix} 3 & 7 & 10 & -15 \\ 3 & -1 & 5 & 0 \\ 66 & 11 & 5 & 1 \\ 0 & -5 & 2 & 2 \end{bmatrix}$$

Hint:
The product of the absolute value of the diagonal terms is
$= |3| \times |-1| \times |5| \times |2| = 30$.

The sum of the absolute value of the terms above the diagonal is
$= |7| + |10| + |-15| + |5| + |0| + |1| = 38$

The value `processedMat` for the above matrix is $= 30 - 38 = -8$.

Module 9: READING FROM AND WRITING TO FILES

Lesson 9.1 – Reading from Files

Learning Objectives

After reading this lesson, you should be able to:

- *read numeric and non-numeric data into MATLAB,*
- *read text (.txt) file contents into MATLAB,*
- *read Excel file contents into MATLAB,*
- *use the concept of a delimiter and commonly used delimiters.*

This lesson begins our discussion on data import and export in MATLAB. For a full list of how to load other data types (such as images, audio, and video) into MATLAB, see the documentation on "Supported File Formats for Import and Export". We will only cover files that contain text data (alphanumeric characters) in this course. We will also focus on text (.txt) and Excel (.xls) files to keep things consistent and simple, but you can review the documentation linked above for specifics on how to use many other common file types.

Why read data from a file?

Applications with large data sets can become laborious and inaccurate if one has to manually input data as variables and arrays into an m-file. Suppose that you are a research engineer working in the materials division of a large company. Your job is to write a program in MATLAB that can determine Young's modulus of a material from a set of stress vs. strain data, which will be collected from an experimental setup. A data acquisition system from the experiment will store hundreds (if not thousands) of stress and strain values in a file. Entering this data into a m-file would be a long and laborious process subject to entry errors. Obviously, it would be much better to have MATLAB read the data directly from the external (it is external to MATLAB) file.

How do I read numeric-only data from files?

There are several ways to read from files in MATLAB. If your file only contains numeric characters, the simplest way to read the data into MATLAB is with dlmread(). It will read the entire contents of the file, which you can store in a numeric variable such as a numeric vector or matrix (as seen in Example 1).

Important Note: The data contained in the file must be numeric in order for `dlmread()` to read the data into MATLAB. Otherwise, MATLAB will throw an error.

Example 1

Read numeric data (shown below from a Notepad window) into MATLAB from a `.txt` file using `dlmread()`. Display to the Command Window the following: all the data, the first row be itself, and the sum of the first two elements in the first row.

Tabs are used as delimiters (or separators) in this file, which we specify as '\t'. Once the data has been read from the file by `dlmread()`, it is automatically put into a matrix, just like we are used to working with, that contains data. We can use all the normal matrix operations on it.

Solution

MATLAB Code example1.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To read numeric-only data from a text file
fprintf('To read numeric-only data from a text file.\n\n')

%------------------------------ INPUTS -------------------------------
%Define the file path where MATLAB should look for the file.
%You need to change the file path to where this file is saved on your
%    computer.
filePath = 'C:\Users\Me\Desktop\numericData.txt';

%------------------------------ SOLUTION -----------------------------
data = dlmread(filePath,'\t');  %Read in tab delimited data from .txt file

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The data read in from ''%s'' is:\n',filePath)
disp(data)

fprintf('Referencing the first row of the data gives us:\n')
firstRow = data(1,:);              %Calling the first row of the matrix
disp(firstRow)

add      = data(1,1) + data(1,2);  %Adding single elements together
fprintf('Adding the first two elements of row one yields %g.\n',add)
```

```
Command Window Output                                          Exam
PURPOSE
To read numeric-only data from a text file.

OUTPUTS
The data read in from 'C:\Users\Me\Desktop\numericData.txt' is:
    3.0000    5.0000   11.0000
   89.0000    0.1000    4.0000
   16.0000   46.0000    9.0000

Referencing the first row of the data gives us:
     3     5    11

Adding the first two elements of row one yields 8.
```

What is a delimiter?

The “delimiter” in a file is what separates one element (number, word, etc.) from another. Delimiters separate the “columns” of data from each other. Similar to the syntax for creating columns in a matrix in MATLAB, common delimiters for text data include tabs (‘\t’), spaces (’ ‘), and commas (’,’). Note the single quotes are required since they are entered as strings. As you can see in the output from Example 1, `dlmread()` automatically puts the data from the file into a matrix format where delimiters define the separate columns and the new lines denote a new row (like hitting “Enter” on the keyboard in the text file) in the output matrix.

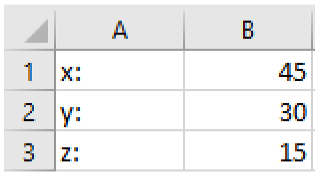
How can I read numeric and character data from files?

The method we use to read the data in the file depends partially on the datatype of the data we want to read. Although `dlmread()`is fast and easy, it cannot handle files that store non-numeric data such as text strings. So, if you have a file that contains any characters, you must use another method.

The method also depends on the type of file where the data is stored. For example, if we want to read data from an Excel file (`.xls`, `.xlsx`, etc.), a good way to accomplish this in MATLAB is with the `xlsread()` function. With `xlsread()`, both numeric and text data can be read while keeping the syntax very simple. Carefully review the use of `xlsread()` in Example 2 as a specific form must be used to have text data returned.

Example 2

Read data (shown below from an Excel window) into MATLAB from an Excel file using `xlsread()`. Display all the data to the Command Window including both numeric and text data.



	A	B
1	x:	45
2	y:	30
3	z:	15

Solution

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To read data from an Excel file
fprintf('To read data from an Excel file.\n\n')

%------------------------------ INPUTS -------------------------------
%Define the file path where MATLAB should look for the file.
%You need to change the file path to where this file is saved on your
%    computer.
filePath = 'C:\Users\Me\Desktop\mixedData.xlsx';

%------------------------------ SOLUTION -----------------------------
%Read data from Excel file
[numData,textData,rawData] = xlsread(filePath);

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The data types for each of the outputs from xlsread() are:\n
fprintf('   numeric data = %s,  text data = %s,  raw data = %s\n\n',.
          class(numData),class(textData),class(rawData))

fprintf('The raw data read in from ''%s'' is:\n',filePath)
disp(rawData)

fprintf('Referencing only the numeric data gives us:\n')
disp(numData)

fprintf('Referencing only the text data gives us:\n')
disp(textData)
```

Command Window Output Example 2

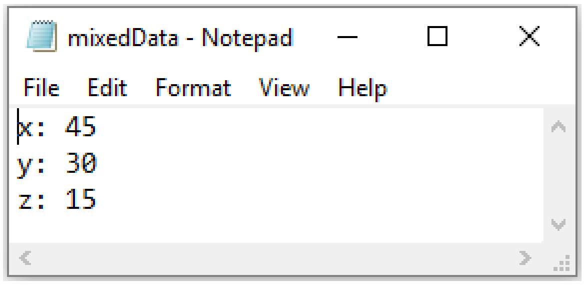
```
PURPOSE
To read data from an Excel file.

OUTPUTS
The data types for each of the outputs from xlsread() are:
   numeric data = double,  text data = cell,  raw data = cell

The raw data read in from 'C:\Users\Me\Desktop\mixedData.xlsx' is:
    'x:'    [45]
    'y:'    [30]
    'z:'    [15]

Referencing only the numeric data gives us:
    45
    30
    15

Referencing only the text data gives us:
    'x:'
    'y:'
    'z:'
```

In the solution, the `xlsread()` function is called with three outputs. This is done so that we can see all of the numeric and character (text) data in the file. Calling `xlsread()` with only one output (e.g., `data = xlsread(filePath)`) will return only the numeric data. For the text and raw data outputs, there are many helpful MATLAB functions to process and reformat the string data contained in the cell arrays. We covered some of these functions in Working with Strings (Lesson 2.3), but there are others that you can investigate if you are working on a similar problem in the future.

Another method we can use to read in a file with both character and numeric data is shown in Example 3, and it uses the function `fgetl()`. The downside of this function is it takes a little more code to read the data into MATLAB, and is explained in Example 3. However, this method demonstrates a more fundamental approach that is, therefore, more universal. Understanding this method can also be useful in debugging errors in file input/output.

Example 3

Read data (shown below from a Notepad window) into MATLAB from a `.txt` file. Display all the data to the Command Window including both numeric and text data. Also find the volume of an object (cuboid) described by the data in the file assuming the numbers describe the dimension of the box along that axis.



```
x: 45
y: 30
z: 15
```

Solution

MATLAB Code example3.m

```matlab
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To read numeric and character data from a file
fprintf('To read numeric and character data from a file.\n\n')

%------------------------------ INPUTS ------------------------------
%Defining the file path where MATLAB should look for the file. You need to
%    change the file path for your computer.
filePath = 'C:\Users\Me\Desktop\mixedData.txt';

%------------------------------ SOLUTION ----------------------------
%We need to tell fopen() whether we are reading ('r'), writing ('w'), or
%    both ('r+').
fileID = fopen(filePath, 'r');

%Creating a loop to read lines of data until an empty line is encountered
%(feof). We need a loop here because fgetl() only reads one line at a time.
lineCount = 1;
while feof(fileID) == 0
    %Read each line of the file one at a time. Returns a string.
    line = fgetl(fileID);

    %Processing the line (string) from the file. Split the string by
    %    whitespace (default delimiter setting) and store in a cell array.
    fileContents(:,lineCount) = split(line);

    %Count how many lines have been read.
    lineCount = lineCount + 1;
end
%These lines go outside the loop because the loop needs to finish before
%   all the data has been read
dimensions = str2double(fileContents(2,:)); %Convert the second row to a
                                             %    numeric datatype
volume     = dimensions(1)*dimensions(2)*dimensions(3); %Volume of the box

%Close the file so it can be used by another process or program
fclose(fileID);

%------------------------------ OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
disp('The contents of the file are:')
disp(fileContents)

fprintf('The volume of the box is %g.\n',volume)
```

```
Command Window Output                                                    Exam
PURPOSE
To read numeric and character data from a file.

OUTPUTS
The contents of the file are:
    'x:'    'y:'    'z:'
    '45'    '30'    '15'

The volume of the box is 20250.
```

To open the file for reading, we use `fopen()`. To watch for the end of the file, we can use `feof()`, which checks the next line from the current one to see if there is any data. If `feof()` does not see any data on the next line, it will return a 1. Otherwise, it returns 0. Finally, we use `fclose()` to close the file we are reading. We did not need to do this for `dlmread()` or `xlsread()` because those functions close the file automatically when they are done reading.

`fgetl()` returns a string. If we want to reformat the line, to work with numeric data for example, we can use `split()` to separate the pieces of the string. From there, we can use the `str2double()` or similar to convert to the desired data type (see Data Types (Lesson 2.5) for more details on conversion).

Lesson Summary of New Syntax and Programming Tools

Task	**Syntax**	**Usage**
Read a file containing numeric-only data into MATLAB	`dlmread()`	`dlmread(myPath,'\t')`
Read an Excel file containing numeric and character data into MATLAB	`xlsread()`	`[num,text] = xlsread(myPath)`
Open a file for reading from or writing to with MATLAB	`fopen()`	`fileID = fopen(myPath, 'r')`
Close a file MATLAB has opened	`fclose()`	`fclose(fileID)`
Check to see if the end of an open file has been reached	`feof()`	`feof(fileID)`
Read a line from a file with MATLAB	`fgetl()`	`fgetl(fileID)`

Task	**Syntax**	**Usage**
Split a string	`split()`	`split(strLine)`
Convert a string to a double data type	`str2double()`	`str2double(var)`

Multiple Choice Quiz

(1). The `dlmread()` function can read files containing

(a) numeric-only

(b) character-only

(c) numbers and characters

(d) None of the above

(2). The `xlsread()` function can read files containing

(a) numeric-only

(b) character-only

(c) numbers and characters

(d) None of the above

(3). The function to open a file for reading is

(a) `fgetl()`

(b) `fopen()`

(c) `fprintf()`

(d) `open()`

(4). To read a line of an external text file, the function is

(a) `fgetl()`

(b) `fopen()`

(c) `read()`

(d) `fprintf()`

(5). You are asked to open the file, ‘problem3.txt’, for reading. Which is the correct choice to complete the following code (your choice will be added at the end of the given m-file)?

(a) `open(file)`

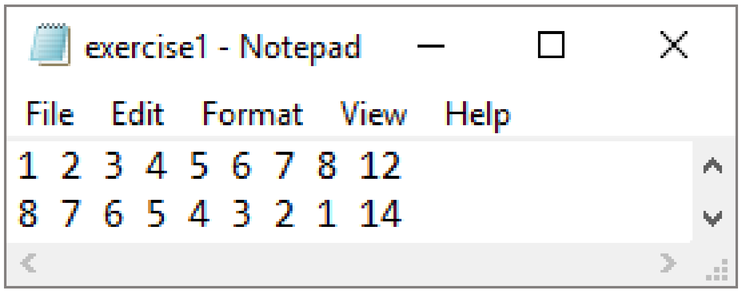
(b) `fopen(file,'r')`

(c) `fopen(file,'w')`

(d) `fopen(file,'o')`

Problem Set

(1). Make a text file as shown in Figure A.



```
1 2 3 4 5 6 7 8 12
8 7 6 5 4 3 2 1 14
```

Figure A: Data file for Exercise 1.

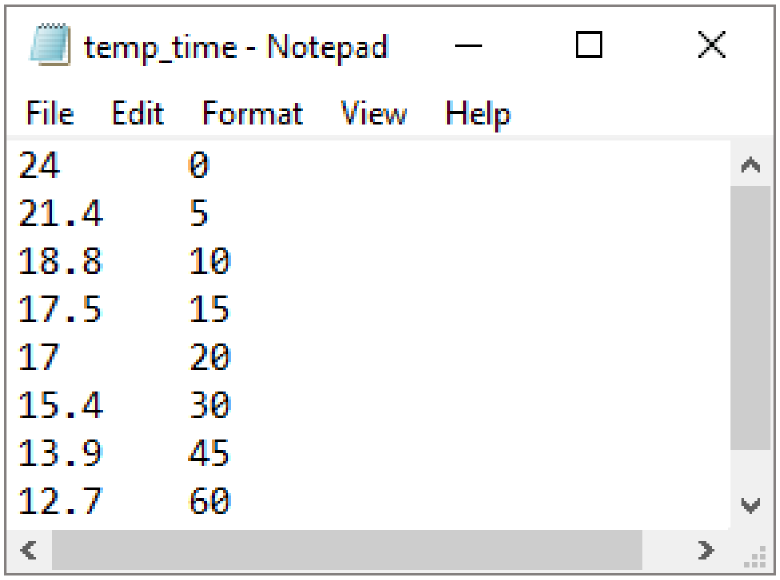
Using MATLAB, write a program that reads the above data file and sums all of the numbers in the first and second rows separately (you should have two sums). Multiply the sum of the first row by the sum of the second. Display the input vectors, and the output product in the Command Window using fprintf() and/or disp().

(2). Read the data from the text file into MATLAB and store it in a matrix. The first column of [A] contains the x data and the second column contains the y data. Plot the data on a 2D plot and label each of the two axes.

$$[\mathrm{A}] = \begin{bmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \\ 5 & 25 \\ 6 & 36 \end{bmatrix}$$

(3). A fuel cell on a satellite is used to power a servomotor. The fuel cell heats up while it is powering the motor. To measure the cooldown time, the interior temperature of the fuel cell is recorded once it is turned off. The text document in Figure B provides the temperature and corresponding time data points. The first column is the temperature (Celsius) of the cell, and the second is the corresponding time (sec).

Read the text file in Figure B by an m-file into two vectors to store the data (temp and time). Plot the temperature of the fuel cell vs. time. Use appropriate figure title and axis labels.



```
24      0
21.4    5
18.8    10
17.5    15
17      20
15.4    30
13.9    45
12.7    60
```

Figure B: Fuel cell temperature and time measurements.

(4). To find contraction of a steel cylinder, one needs to regress the thermal expansion coefficient data to temperature. The data is given below.

Table D: The thermal expansion coefficient of steel at given temperatures.

Temperature, T, (°F)	Coefficient of thermal expansion, α, (in/in/°F)
80	6.47 * 10^-6
40	6.24 * 10^-6
-40	5.72 * 10^-6
-120	5.09 * 10^-6
-200	4.30 * 10^-6
-280	3.33 * 10^-6
-340	2.45 * 10^-6

Put the above data in a two-column format in a text file.

(a) Read the above data into two vectors, temp and alphaVal.

(b) Fit the above data to by using the polyfit() function. Plot the regression model along with the data points. Use proper axis labels, title, and legend in the figure.

(c) Find the predicted value of the coefficient of thermal expansion at T = -100°F.

(d) If the steel cylinder is dipped in a dry-ice/alcohol bath, the diametric contraction, D, in the steel cylinder is given by

$$\Delta D = D\int_{T_r}^{T_f} \alpha dT$$

where,

D = outer diameter of the cylinder,
T_r= room temperature,
T_f= dry-ice/alcohol temperature

Find the diametric contraction.

(e) The contraction obtained in part (d) is not adequate, as specifications require a diametric contraction of at least 0.02". Find the temperature of the cooling medium you would need to achieve that much contraction.

(f) Find the rate of change of diametric contraction at and. What do you conclude from these results?

(5). Write a MATLAB program that reads the input text file given in Figure C, and outputs (to the Command Window) the longitudinal Young's modulus of the composite material. To find the Young's modulus, you may use the following regression model

$$E = \frac{\sum_{i=1}^{n} \sigma_i \varepsilon_i}{\sum_{i=1}^{n} (\varepsilon_i)^2}$$

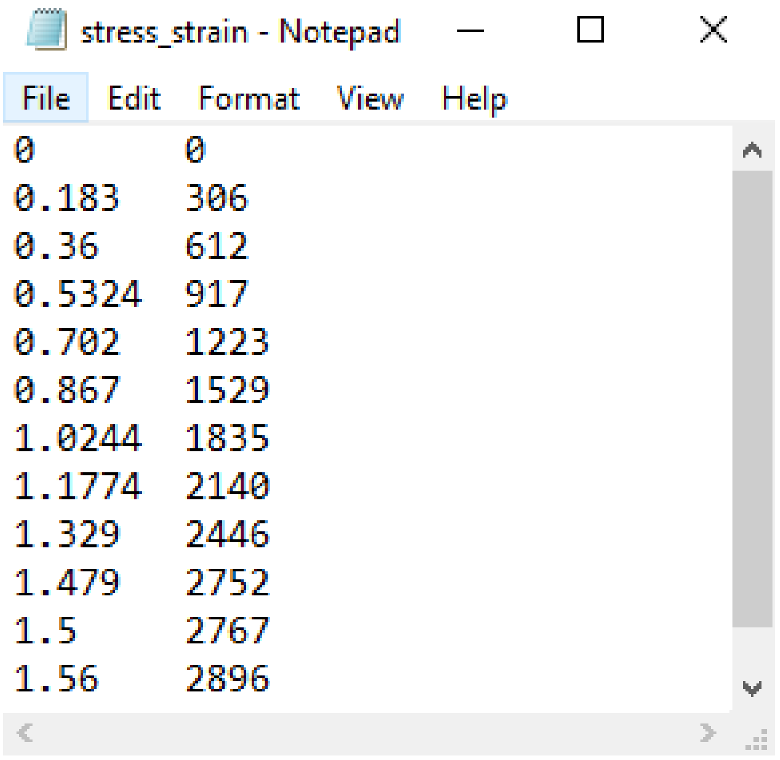
where,

E is Young's modulus (Pa),
ε is the strain (m/m),
σ is the stress (Pa),

n is the number of data points.

A tensile test of a composite material has provided the stress-strain data that is given in Figure C. The strain (cm/cm) and stress (MPa) are given in the first and second columns, respectively.

Use the `fprintf()` function with `%e` format to display your output in the Command Window. Be sure to note the units.



```
0         0
0.183     306
0.36      612
0.5324    917
0.702     1223
0.867     1529
1.0244    1835
1.1774    2140
1.329     2446
1.479     2752
1.5       2767
1.56      2896
```

Figure C: Strain and stress values of a composite material.

Module 9: READING FROM AND WRITING TO FILES

Lesson 9.2 – Writing to Files

Lesson Objectives

After reading this lesson, you should be able to:

- *write numeric data to text files using MATLAB,*
- *write numeric data to Excel files using MATLAB,*
- *write non-numeric data to text files using MATLAB,*
- *write non-numeric data to Excel files using MATLAB,*
- *apply the concept of a delimiter when writing data.*

How can I write numeric data to files with MATLAB?

Many of the concepts we learned when reading from files in Lesson 9.1 using `dlmread()` are also applicable when we use `dlmwrite()` for writing to a file. `dlmwrite()` is not the only method for writing to files, but it is one of the most straightforward. Similarly to `dlmread()`, `dlmwrite()` can only handle numeric data. Also, although we focus on writing to text files here, `dlmwrite()` is also compatible with the common .csv (comma separated values) file format.

Example 1

Write the numeric-only data given in matrix $[A]$ to a file called '`numeric_data.txt`' using `dlmwrite()`.

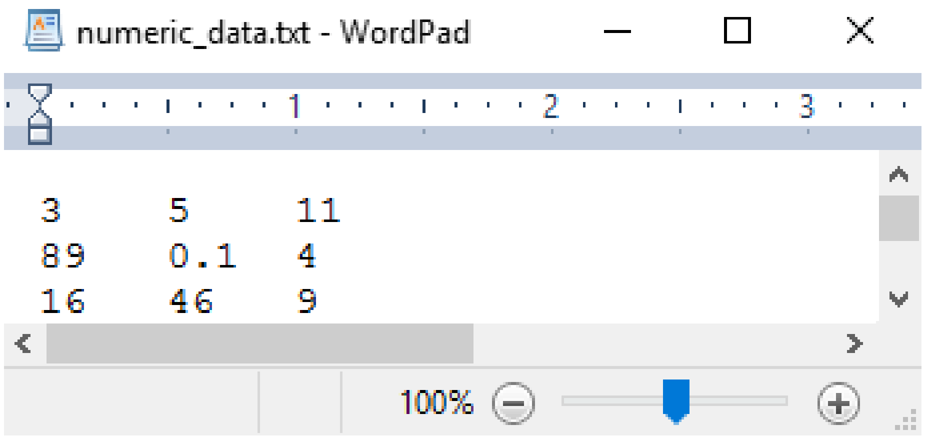
$$[\mathrm{A}] = \begin{bmatrix} 10 & 5 & 11 \\ 89 & 0.1 & 4 \\ 16 & 46 & 9 \end{bmatrix}$$

Solution

```
MATLAB Code                                                          examp
```
```
clc
clear

%------------------------------ PURPOSE ------------------------------
%To write numeric data to a text file
```

```
MATLAB Code (continued)                                              examp
```
```
%------------------------------ INPUTS ------------------------------
%Defining the file path where MATLAB should look for the file.
%You need to change the file path for where the file is on your compu
filePath = 'C:\Users\Me\Desktop\numeric_data.txt';

%------------------------------ SOLUTION ------------------------------
%Defining some data we want to write to file.
matrix = [3  5   11;
          89 0.1 4;
          16 46  9];

%For dlmwrite(), all data must be numeric.
%Tabs are use as delimiters in the file, which we specify as '\t'.
dlmwrite(filePath,matrix,'\t')
```

Note, the generated file shown in Figure 1 is opened in WordPad. This is because NotePad does not recognize end-of-line characters and would not properly display the matrix without adding some additional parameters to `dlmwrite()`. This is a minor detail and not something to be concerned about.



```
3     5     11
89    0.1   4
16    46    9
```

Figure 1: The opened file that contains the data we wrote to it in Example 1. You can use WordPad or equivalent program to create/edit `.txt` files.

We do not need to tell `dlmwrite()` to create a file as it will do so automatically if the file does not already exist. If a file *does* exist, it will overwrite the data contained in the file unless the append parameter is used (see documentation). Likewise, it will open and close the file automatically.

How can I write non-numeric data to files with MATLAB?

To write mixed data types (for example, numbers and strings), we need a way to store the data we want to write, and we need a write (or "print") function that can handle both numeric and nonnumeric data. Previously, we used `fprintf()` to print messages to the Command Window, but now we will use it to write data to a text (`.txt`) file. The syntax and function of `fprintf()` are almost exactly the same in this application except we now need to give it the fileID, which is a unique identifier for that file (in case multiple files are open), to write to as seen in Example 2. You can see the lesson on Reading from Files (Lesson 9.2) for details on why we need `fopen()` and `fclose()`.

Example 2

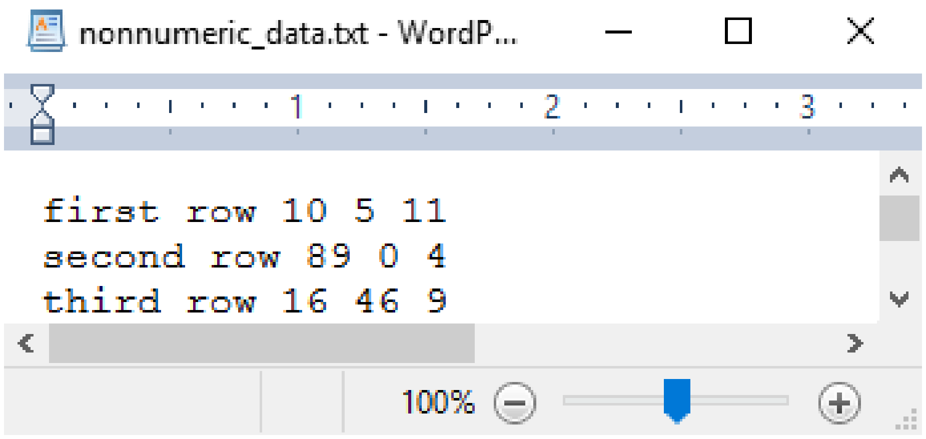
Write non-numeric (strings) and numeric (numbers) data to a file named '`nonnumeric_data.txt`'. Write the data given in the matrix $[A]$ to the file. The first column of the file should be a label for each row in the form "first row", "second row", etc. For example, the first row written to the text file should read "first row 10 5 11".

$$[A] = \begin{bmatrix} 10 & 5 & 11 \\ 89 & 0.1 & 4 \\ 16 & 46 & 9 \end{bmatrix}$$

Solution

MATLAB Code examp

```
clc
clear

%------------------------------- PURPOSE ------------------------------
%To write non-numeric data to a text file

%------------------------------- INPUTS -------------------------------
%Opening the text file we want to read from
%Defining the file path where MATLAB should look for the file. You ne
%    change the file path for your computer.
filePath = 'C:\Users\Me\Desktop\nonnumeric_data.txt';

%------------------------------- SOLUTION -----------------------------
%We need to tell fopen() whether we are reading ('r'), writing ('w'), o
%     ('r+').
fileID = fopen(filePath, 'w');

%We are using the cell array data type because cell arrays can
% store numeric and/or non-numeric values in the same array
arrayToWrite = {'first row', 10, 5, 11;
                'second row', 89, 0.1, 4;
                'third row', 16, 46, 9};

%Writing the data to the file one row at a time in a loop
for row = 1:size(arrayToWrite,1)
    %This line writes to the file rather than printing to the Command
    fprintf(fileID,'%s %.0f %.0f %.0f\n',arrayToWrite{row,:});
end

%Closing the file after we have finished our process
fclose(fileID);
```



```
first row 10 5 11
second row 89 0 4
third row 16 46 9
```

Figure 2: The opened file that contains the data we wrote to it in Example 2.

You can use Notepad, WordPad, or equivalent program to create/edit `.txt` files. As you can see in the opened text file in Figure 2, each row of the data is written to a line in the text file. In Example 2, all the strings are in the first column, so we knew which columns would contain only strings and which only numbers.

In some cases, you may want to write data in a more readable format such as to an Excel spreadsheet using `xlswrite()` or to a table format using `writetable()`. Example 3 shows how to write data to an Excel file using the `xlswrite()` function. As mentioned in Lesson 9.1, you can review MATLAB documentation on "Supported File Formats for Import and Export" for a quick reference on all the different options.

Example 3

Write non-numeric (strings) and numeric (numbers) data to an Excel file named '`mixedData.xlsx`'. Write the data given in matrix $[A]$ to the file. The first column of the file should be a label for each row in the form "first row", "second row", etc. For example, the first row written to the text file should read "first row 10 5 11".

$$[\mathrm{A}] = \begin{bmatrix} 10 & 5 & 11 \\ 89 & 0.1 & 4 \\ 16 & 46 & 9 \end{bmatrix}$$

Solution

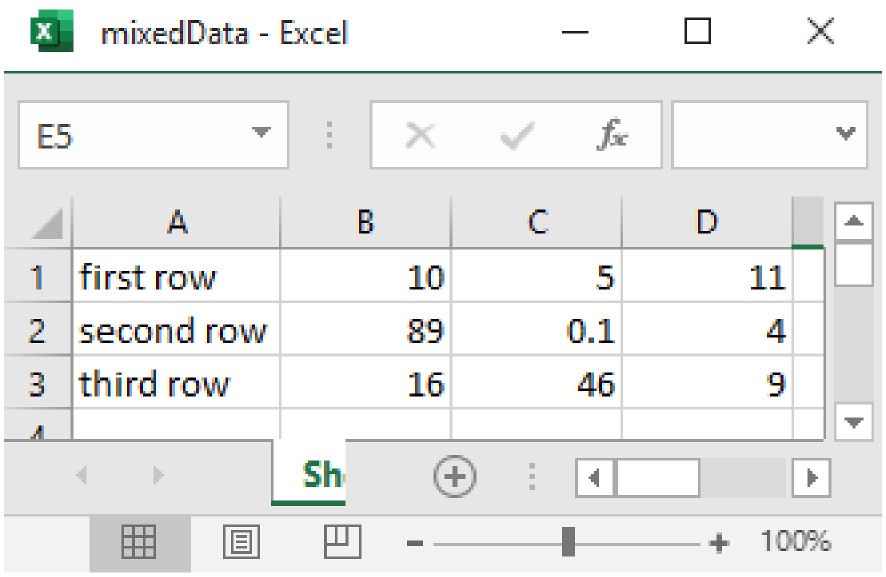
MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
%To write data to an Excel file

%------------------------------ INPUTS -------------------------------
%Defining the file path where MATLAB should look for the file.
%You need to change the file path for where the file is on your compu
filePath = 'C:\Users\Me\Desktop\mixedData.xlsx';
```

MATLAB Code (continued) examp

```
%------------------------------ SOLUTION -----------------------------
%Defining some data we want to write to file.
arrayToWrite = {'first row', 10, 5, 11;
                'second row', 89, 0.1, 4;
                'third row', 16, 46, 9};

%Write data to an Excel file using xlswrite()
xlswrite(filePath,arrayToWrite)
```

As seen in Figure 3, the rows and columns of the cell array in MATLAB are written as rows and columns in the Excel spreadsheet.



	A	B	C	D
1	first row	10	5	11
2	second row	89	0.1	4
3	third row	16	46	9

Figure 3: The opened Excel file that contains the data we wrote to it in Example 3.

Lesson Summary of New Syntax and Programming Tools

Task	Syntax	Usage
Write to an Excel file	xlswrite()	xlswrite(filePath,array)
Write to a text file	dlmwrite()	dlmwrite(filePath,array)

Multiple Choice Quiz

(1). The `dlmwrite()` function can write data containing

(a) numeric-only

(b) character-only

(c) numbers and characters

(d) None of the above

(2). Appending data to a file means MATLAB will

(a) Erase all the data in the file and then write the new data

(b) Add the new data to the beginning of the file and leave existing data intact

(c) Add the new data to the end of the file and leave existing data intact

(d) do nothing because MATLAB cannot append data

(3). To open a file for writing, the correct choice is

(a) `fopen(filePath)`

(b) `fopen(filePath,'r')`

(c) `fopen(filePath,'w')`

(d) `fopen(filePath,'o')`

(4). To write a line to an external text file, the most appropriate function is

(a) `fgetl()`

(b) `fopen()`

(c) `read()`

(d) `fprintf()`

(5). Complete the code to output the variable, a, to the text file, fo.

(a) `fprintf(fwrite,'The number is %g',a)`

(b) `fprintf('The number is %g',a)`

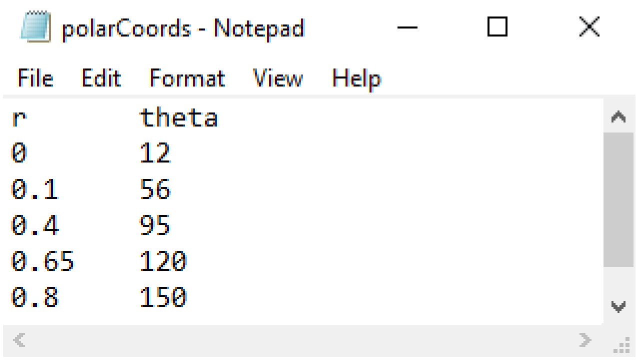
(c) `fprintf(fwrite,'The number is %g')`

(d) `disp(fwrite,a)`

Problem Set

(1). Use the `rand()` function to generate some "sensor readings". Assume our pseudo-sensor reports location in 3D space, and create "readings" for each axis (x, y, z). Pick a range that makes sense to you and generate at least 100 numbers for each variable. Write each of these variables to a text file (.txt) and an Excel file (.xlsx). The first column in the file should be the time associated with the sensor reading. Assume time starts at 0 and readings are taken every 0.01 seconds. The first row in the file should have a label for each row: namely, "time", "x", "y", "z".

(2). Create the text file shown below in Figure A, which contains polar coordinates (r, θ). Read the file into MATLAB, convert the polar coordinates to Cartesian coordinates (x, y), and write the Cartesian coordinates to an Excel file called 'cartCoords.xlsx'. The first row should be the labels for each column ("x" and "y").



```
r       theta
0       12
0.1     56
0.4     95
0.65    120
0.8     150
```

Figure A: A text file containing polar coordinate data.

(3). The Young's modulus was calculated from discrete data in Lesson 9.1 Exercise 3. Append the text file that was used in Lesson 9.1 Exercise 3 with the calculated Young's modulus and a description (e.g., "The Young's modulus is ...") with appropriate units. Be sure not to overwrite the original data contained in the text file.

(4). Write matrix [D], given below, to a text file. In the same file, on separate lines, write:

- "The size of the matrix is ..."
- "The norm of the matrix is ..."
- "The trace of the matrix is ..."

$$[\mathrm{D}] = \begin{bmatrix} 16 & 2 & 3 \\ 5 & 11 & 10 \\ 9 & 7 & 6 \end{bmatrix}$$

Hint: To avoid hardcoding, you use loops to write the matrix such that your solution will work for a file containing any number of rows. Assume that [D] will be of size $3 \times$ m (i.e., it will always have three columns).

(5). Using the solution from Lesson 9.1 Exercise 1, append the sum of each row to the same m-file ('exercise1.txt') in the form "The sum of row X is ...". Use loops such that your solution will work for a file containing any number of rows.

Module 9: READING FROM AND WRITING TO FILES

Lesson 9.3 – Navigating Directories in MATLAB

Lesson Objectives

After reading this lesson, you should be able to:

- *change the current directory,*
- *make directories (folders),*
- *remove directories (folders),*
- *loop through directory structures.*

How do I set the current working directory for MATLAB?

Setting the working directory/folder in MATLAB means you are telling MATLAB where you want it to look for and work with files. That is, the working directory is the directory/system folder from which you are currently working. The current directory is usually set to the location of your last active m-file (the last one you ran). To view or output your current working directory, use the `pwd` ("print working directory") command. To change the working directory, use the `cd` ("change directory") command.

You can use the function `dir()` to create a directory struct array. This struct array will then have file attributes (information) such as names or size for that directory, which can be referenced via dot notation (e.g., myFolder.name). `dir()` is useful if your m-file is in a different directory (folder) than the data files you want to access, which we will demonstrate in Example 2.

Example 1

Create a struct variable containing a directory's information. Display the name, bytes, and isdir fields to the Command Window.

Solution

Command Window Output Exam

```
PURPOSE
To create a struct variable from directory information.

OUTPUTS
dataLocation =
  7×1 struct array with fields:

    name
    folder
    date
    bytes
    isdir
    datenum


dataTable =
  7×3 table

           name            bytes    isdir
    __________________     _____    _____

    '.'                      0      true
    '..'                     0      true
    'Category A'             0      true
    'Category B'             0      true
    'Category C'             0      true
    'Category D'             0      true
    'example_file.txt'       0      false
```

MATLAB Code (continued) example1.m

```
%------------------------------ INPUTS ------------------------------
%Defining file path
filePath = 'C:\Users\Me\Desktop\Example Directory';

%------------------------------ SOLUTION ----------------------------
%Saving directory of desired location as a struct array variable
dataLocation = dir(filePath);

%------------------------------ OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
%Displaying fields of the struct array (dataLocation)
dataLocation

%Reformatting into something more informative (a table)
dataTable = struct2table(dataLocation);  %Unsuppress this to see original
dataTable = dataTable(:,{'name','bytes','isdir'})
```

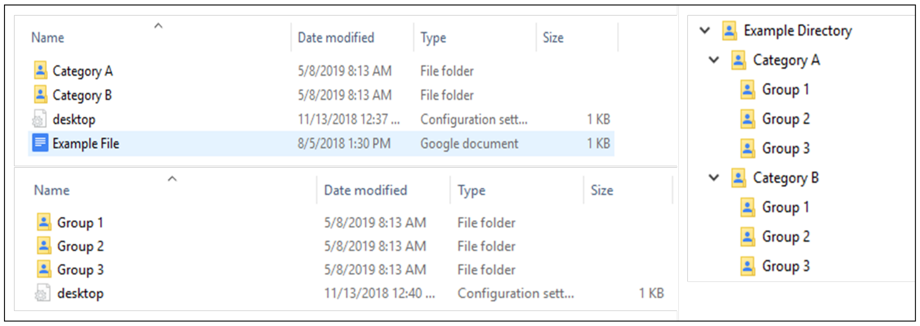
Command Window Output Exam

```
PURPOSE
To create a struct variable from directory information.

OUTPUTS
dataLocation =
  7×1 struct array with fields:

    name
    folder
    date
    bytes
    isdir
    datenum


dataTable =
  7×3 table

           name            bytes    isdir
    __________________     _____    _____

    '.'                      0      true
    '..'                     0      true
    'Category A'             0      true
    'Category B'             0      true
    'Category C'             0      true
    'Category D'             0      true
    'example_file.txt'       0      false
```

The first output we see from Example 1 are the file attributes contained in the struct array (`dataLocation`) that we created using `dir()`. Next, we can see the outputs from the name, bytes, and isdir fields of the struct array (`dataLocation`) in table form. For the list of names, notice the first two entries are "." and "..", which are hidden folders in Windows. If you are on macOS watch out for ".DS_Store" hidden files. That means you will not be able to see them if you look in that directory (see Figure 2) unless you have specifically told Windows to show you those hidden folders. Most of the time, you do not have to worry about or deal with hidden files and folders. The exception is when you want to loop through all of a directory's contents.

The directory structure used in Examples 1, 2, and 3 (see Figure 2) has a parent directory called "Example Directory" (a folder containing all the data).

Within the parent directory, there is a folder for "Category A" and one for "Category B". Finally, within each "Category", there are three group folders: "Group 1", "Group 2", and "Group 3". An example file path would be: "*C:\Users\MechPlus\Example Directory\Category A\Group 2\file1.txt*".



Name	Date modified	Type	Size
Category A	5/8/2019 8:13 AM	File folder	
Category B	5/8/2019 8:13 AM	File folder	
desktop	11/13/2018 12:37 ...	Configuration sett...	1 KB
Example File	8/5/2018 1:30 PM	Google document	1 KB

Name	Date modified	Type	Size
Group 1	5/8/2019 8:13 AM	File folder	
Group 2	5/8/2019 8:13 AM	File folder	
Group 3	5/8/2019 8:13 AM	File folder	
desktop	11/13/2018 12:40 ...	Configuration sett...	1 KB

```
Example Directory
  Category A
    Group 1
    Group 2
    Group 3
  Category B
    Group 1
    Group 2
    Group 3
```

Figure 2: File structure used in Examples 1, 2, and 3. On the left, you can see the folder contents view. On right, is a tree view for the same directories.

How can I loop through the contents of a directory?

Sometimes you will have to look through and extract data from many files in many directories. To automate the process of looking through and opening files and folders, we can use the name field of a directory struct array, which gives the user the folder/file name of the item. Note that this field itself is an array with a size equal to the number of items contained in the directory. For instance, in Example 1, the field was an array of six elements (names). Example 2 demonstrates how to use this field to loop through any directory's contents without knowing the contents' folder/file names.

In our case, we have three directory levels to loop through ("Category", "Group", and the files in each group). To access all of the files and folders contained in each category, we will need three loops: one for each level. We mimic the structure of the directory by nesting the loop for "Group" within the loop for "Category" and the loop for the files in each group within the loop for "Group".

Important Note: If you are using MATLAB on Windows, you must account for any hidden files and folders when automatically searching/looping through directories.

Example 2

Loop through and read all the files contained in the directories shown in Figure 2.

Solution

MATLAB Code example2.m

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To loop through the contents of a directory structure
fprintf('To loop through the contents of a directory structure.\n\n')

%------------------------------ INPUTS -------------------------------
%Defining file path to directory
filePath = 'C:\Users\Me\Desktop\Example Directory';

%------------------------------ SOLUTION -----------------------------
categoryDir = dir(filePath);  %Saving directory file path as a struct array

data = {};  %Initializing empty cell array to store lines of data that we
            %   read from the files.
row = 1;    %Data storage counter index

%This loop that will handle looping through all the “Category” folders
%(Windows users) Skip the first two entries since they are hidden files
%    (Note: Loop starts at 3, not 1)
for categories = 3:length(categoryDir)
    currentCategory = categoryDir(categories).name;
    groupDir = dir([filePath '\' currentCategory]);

    %This loop that will handle looping through all the “Group” folders
    for group = 3:length(groupDir)
        currentGroup = groupDir(group).name;
        fileDir = dir([filePath '\' currentCategory '\' currentGroup ...
                        '\' '*.txt']);

        for files = 1:length(fileDir)
            currentFile = fileDir(files).name;
                %Reading the contents of each file and storing them in a
                %  cell array. We need a cell array because they are
                %  strings of differing lengths.
                data{row} = dlmread([filePath '\' currentCategory '\'...
                                      currentGroup '\' currentFile]);

                row = row + 1;  %Incrementing the data storage counter
        end
    end
end
```

MATLAB Code (continued) examp

```
%----------------------------- OUTPUTS ------------------------------
fprintf('OUTPUTS\n')
fprintf('The contents of the whole cell array are:\n')
disp(data)
fprintf(['The contents of first file that we opened and read',...
         ' (Category A, Group 1):\n'])
disp(data{1})
fprintf(['The contents of second file that we opened and read',...
        ' (Category A, Group 2):\n'])
disp(data{2})
```

Command Window Output Exam

```
PURPOSE
To loop through the contents of a directory structure.

OUTPUTS
The contents of the all files (whole cell array) are:
  Columns 1 through 4

    [1×2 double]    [1×2 double]    [1×2 double]    [1×2 double]

  Columns 5 through 6

    [1×2 double]    [1×2 double]

The contents of first file that we opened and read (Category A, Group
     1     1

The contents of second file that we opened and read (Category A, Grou
     1     2
```

Another way to ignore unwanted files and/or folders is to use the wildcard character, *. You can see this in Example 2 where we ignore files that are not text (`.txt`) files. The wildcard character can be used at the beginning or in the middle of a file/folder name. For example, "trial*.txt" would include all files that begin with "trial" and end with ".txt".

To make Example 2 a little more compact, we could ignore the hidden files using the wildcard character with the `dir()` function. Try editing Example 2 to accomplish this yourself. Remember to start the loop from 1 in that case.

Note that the solution given in Example 2 will always ignore the hidden folders. This solution can be useful if you have many different folder names at each level, which would make using the wildcard character less efficient.

How can I create a new folder (directory) with MATLAB?

We saw in Lesson 9.2 that MATLAB will create a file to write data to if it does not exist. However, MATLAB can also create directories (folders) for you using the command `mkdir`; though, it does not do this automatically in most cases. The inverse (removing a folder) is accomplished with `rmdir`.

Important Note: Be careful using `rmdir` to remove directories. Removing a directory using `rmdir` will not send the directory/folder and its contents to the system trash bin and cannot be recovered!

Example 3

Make a new folder called Category D in the directory shown in Figure 2 using MATLAB.

Solution

MATLAB Code examp

```
clc
clear

%------------------------------ PURPOSE ------------------------------
fprintf('PURPOSE\n')
%To create a new folder with MATLAB
fprintf('To create a new folder with MATLAB.\n\n')

%------------------------------ INPUTS -------------------------------
%Defining parent folder path
filePath  = 'C:\Users\Me\Desktop\Example Directory';

%------------------------- SOLUTION/OUTPUTS --------------------------
fprintf('OUTPUTS\n')
dataLocation = dir(filePath);
%Converting the struct output into a table (more readable format)
%    Skip the first two entries since they are hidden files
dataTable = struct2table(dataLocation(3:end));
%Only want to look at two columns to keep things simple
dataTable = dataTable(:, {'name','folder'});

fprintf('***Contents BEFORE creating new folder.***\n')
dataTable

mkdir(filePath, 'Category D')
%Checking the contents of the directory after creating new folder\
fprintf('\n***Contents AFTER creating new folder.***\n')
dataTable
```

```
Command Window Output                                              Example 2
PURPOSE
To loop through the contents of a directory structure.

OUTPUTS
The contents of the all files (whole cell array) are:
  Columns 1 through 4

    [1×2 double]    [1×2 double]    [1×2 double]    [1×2 double]

  Columns 5 through 6

    [1×2 double]    [1×2 double]

The contents of first file that we opened and read (Category A, Group 1):
     1     1

The contents of second file that we opened and read (Category A, Group 2):
     1     2
```

Lesson Summary

Task	Syntax	Usage
Print working directory	`pwd`	`pwd`
Change current directory	`cd`	`cd 'myPath'`
Create a directory struct array	`dir()`	`dir('myDirectoryPath)`
Create a new folder	`mkdir`	`mkdir 'New Folder'`
Remove an existing folder	`rmdir`	`rmdir 'Folder'`

Multiple Choice Quiz

(1). The `pwd` command

(a) returns the MATLAB working directory

(b) returns the password for your MathWorks account

(c) returns the password for your Windows account

(d) erases the given directory

(2). The `dir()` function does not return

(a) the names of the items in a directory

(b) the sizes of the items in a directory

(c) the data contained in files in a directory

(d) whether the item is a directory or not

(3). The `dir()` function creates a ____________ variable that contains information about the contents of the directory.

(a) struct

(b) string

(c) numeric

(d) logical

(4). The `rmdir` command

(a) remakes the given directory (erases the contents, but leaves the > folder)

(b) moves the directory to the Recycle Bin (in Windows)

(c) permanently deletes the directory

(d) makes a new folder inside the current directory

(5). The `cd` command can change the location of the

(a) install directory of MATLAB

(b) executed m-file

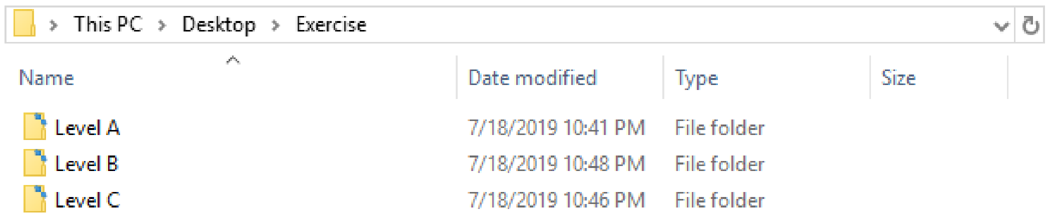
(c) current directory

(d) None of the above

Problem Set

(1). In this exercise, you will create the necessary folders and files to test the solutions you write in exercises 2-6.

Recreate the directory (folders) shown in Figure A where there are two levels of folders. Two levels mean one parent folder (like "Exercise 1") and one level of subfolders called "Level A", "Level B", "Level C".

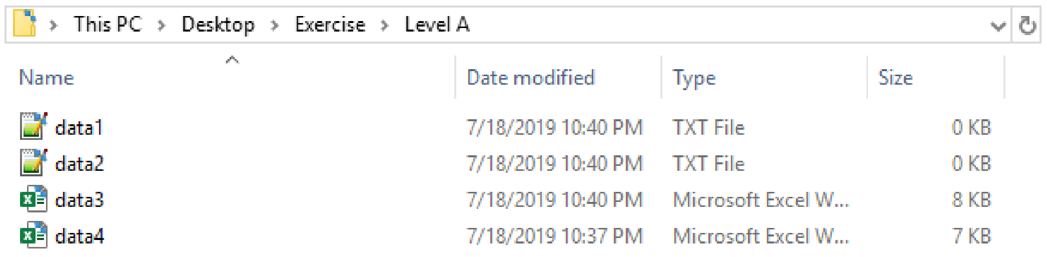


› This PC › Desktop › Exercise

Name	Date modified	Type	Size
Level A	7/18/2019 10:41 PM	File folder	
Level B	7/18/2019 10:48 PM	File folder	
Level C	7/18/2019 10:46 PM	File folder	

Figure A: Directory structure for use in the excises.

Within each subfolder ("Level *"), you will need to create some files. This should look like those files shown Figure B: two text files and two Excel files inside of each "Level". In some exercises, you will only read text files or only Excel files. You can easily account for this in your solution using a conditional statement or wildcard character, which were techniques shown and described in this lesson.

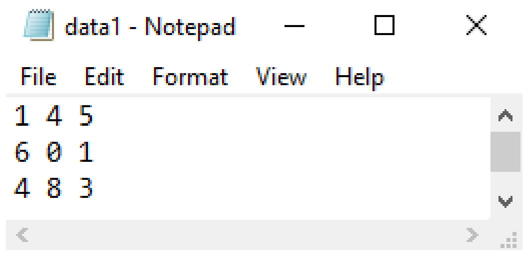


› This PC › Desktop › Exercise › Level A

Name	Date modified	Type	Size
data1	7/18/2019 10:40 PM	TXT File	0 KB
data2	7/18/2019 10:40 PM	TXT File	0 KB
data3	7/18/2019 10:40 PM	Microsoft Excel W...	8 KB
data4	7/18/2019 10:37 PM	Microsoft Excel W...	7 KB

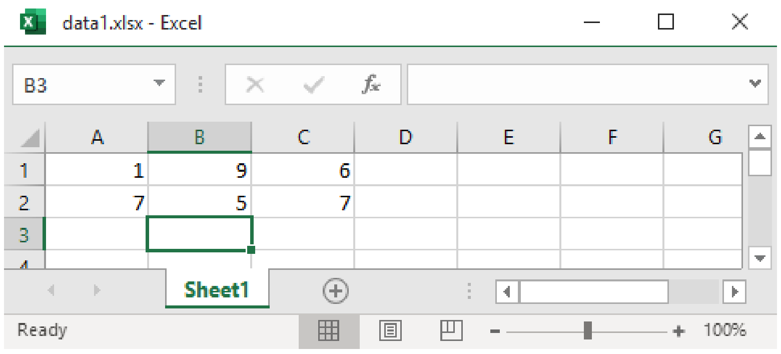
Figure B: An example of what the contents of each "Level" folder should be.

Within each file that you create, you will need to enter some data to read. Make the data in each file unique, so each file is identifiable from the data. You can quickly and easily do this by creating the numeric contents of the files using `magic()` or `rand()` in MATLAB and then copying those numbers to the files. Each exercise will specify what type of data the files should contain (numeric, non-numeric, or mixed). Examples of what text (.txt) and Excel (.xlsx) files should look like are given in Figures C and D.



```
1 4 5
6 0 1
4 8 3
```

Figure C: An example of what the contents of a text (.txt) file might look like.



	A	B	C	D	E	F	G
1	1	9	6				
2	7	5	7				
3							

Figure D: An example of what the contents of an Excel (.xlsx) file might look like.

Note: For all the exercises given below, you should write your solutions without hardcoding the names of any of the folders/files you are reading from. Use the appropriate MATLAB syntax to read all the files no matter what the folder/file names are.

(2). ***Task:*** Make a struct variable for a folder/directory using `dir()` and display info (fields) for that folder/directory in the Command Window.

Testing: To test this program, input the folder path to one of the folders you created in Exercise 1 or any other folder.

(3). ***Task:*** Create a MATLAB function using, in part, your solution from Exercise 2. The function should read data in from the directory (folder) structure with two levels (see explanation in Exercise 1).

The function should read data for all *Excel files* contained in the folders where the Excel files can contain numbers, text, or both types of data. Display the data in the Command Window.

Testing: To test your program, use the folders and files you created in Exercise 1. Make the contents of the Excel files be labels in the first row and numbers in all the following rows.

(4). ***Setup:*** Using the function you created in Exercise 3, read all the data contained in *Excel files* from any directory structure with two levels (see explanation in Exercise 1). You are told the files contain only numeric data.

Task: Find the sum of each row in each file. Store these sums in a vector in MATLAB, and output the vector to the Command Window.

Testing: To test your program, use the folders and files you created in Exercise 1. Make the contents of each Excel file be an n × n matrix.

(5). ***Setup:*** Modify the function you created in Exercise 3 to read all the data contained in *text files* from any directory structure with two levels (see explanation in Exercise 1). You are told the files contain only numeric data.

Task: Find the average of each row for the matrix contained in each file, and append the averages to that file.

Testing: To test your program, use the folders and files you created in Exercise 1. Make the contents of each Excel file be an n × n matrix (You can use the same ones from).

(6). ***Setup:*** Modify the function you created in Exercise 3 to read all the data contained in *Excel and/or text files* from any directory structure with two levels (see explanation in Exercise 1). You are told the text file contains only numeric data in two columns (size n × 2).

Task: Plot the data on a 2D plot where the numbers in first column are the x data points and the numbers in the second column are the y data points. There should be one line for each file on the same plot. Make a legend that shows the filename associated with each set of data (plotted line).

Testing: To test your program, use the folders and files you created in Exercise 1. Make the contents of each file be two columns of numbers (size n × 2).