Autar Kaw
Benjamin Rigsby
Daniel Miller
Ismet Handžić

Introduction to Programming Concepts with MATLAB

3rd Edition

Introduction to Programming Concepts with MATLAB

Autar Kaw
Benjamin Rigsby
Daniel Miller
Ismet Handžić

AutarKaw.com

"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.

Front cover design by Ismet Handžić and Benjamin Rigsby

Introduction to Programming Concepts with MATLAB

Third Edition

Autar Kaw, Benjamin Rigsby, Daniel Miller, Ismet Handžić

AutarKaw.com

To Sherrie, Candace, Angelie, and Bucky
AK

To Victoria
BR

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

To my family
IH

TABLE OF CONTENTS

ABOUT THE AUTHORS x
PREFACE xiv

MODULE 1: INITIAL SETUP AND BASIC OPERATION

Lesson 1.1 – MATLAB Introduction 2
What is MATLAB? 2
What is MATLAB used for in engineering and science? 2
How can I get MATLAB onto my computer? 3
Are there any free alternatives to MATLAB? 3
Where can I find more information and help with MATLAB online? 3

Lesson 1.2 – Hello World 4
Where can I find and open the MATLAB program? 4
Step 1: Create a New m-file 4
Step 2: Write the 'Hello World' Code 5
Step 3: Run the Program 6
Step 4: Make Your Program a Little Fancier 6

Lesson 1.3 – MATLAB Environment 8
MATLAB Environment Windows and Parts 8
Navigation Ribbon 9
Working Folder Location 9
Current Folder 10
Command Window 10
Editor Window 11
Workspace 11
Status Bar 12

Lesson 1.4 – Changing MATLAB Preferences 13
How can I change the window layout in MATLAB? 13
Changing Basic User Preferences 14

Lesson 1.5 – The m-file 16
What is an m-file? 16
Where can I create a new m-file? 17
How do I save my m-file? 17
How do I input variables and expressions into the m-file? 17
How do I run the m-file? 18
What are the clc and clear commands? 19
How can I place comments in my m-file? 19
Can I separate my code into parts within the m-file? 20
What does the color highlighting in the m-file mean? 20
Multiple Choice Quiz 22

Exercises 23

Lesson 1.6 – The Command Window 27
What is the Command Window and how can I use it? 27
How can I suppress outputs in the Command Window? 28
Can I view help from the Command Window? 29
Can the Command Window do it all? 30
Multiple Choice Quiz 31
Exercises 32

Lesson 1.7 – Publishing an m-file 33
What does publishing do? 33
How can I publish in MATLAB? 34
How can I get a PDF file of my published code? 34

MODULE 2: BASIC PROGRAMMING FUNDAMENTALS

Lesson 2.1 – Variables and Naming Rules 38
What is a mathematical variable? 38
What is a programming variable? 38
How can I give my results a variable name of their own? 38
What are some possible problems with naming an expression? 40
Are there benefits to good practices for variable naming? 41
Are there some guidelines for variable naming that I can follow? 42
Multiple Choice Quiz 45
Exercises 47

Lesson 2.2 – Characters and Strings 48
What is a character? 48
What is a string? 49
What makes characters/strings special? 50
Multiple Choice Quiz 52
Exercises 54

Lesson 2.3 – Working with Strings 55
How do I join two strings together? 55
How do I search and count strings? 56
How do I make a whole string lower or upper case? 57
Can I split a string into its component pieces? 58
Multiple Choice Quiz 61
Exercises 63

Lesson 2.4 – Inputs and Outputs 64
How can I get an input from the user? 64
How do I display notes in the Command Window? 64
Multiple Choice Quiz 69
Exercises 71

Lesson 2.5 – Data Types 73
What is a data type? 73

What are the MATLAB data types? 73
Why are data types important? 74
How do I check the data type of a variable? 74
Can I convert between data types? 76
Multiple Choice Quiz 80
Exercises 81

Lesson 2.6 – Vectors and Matrices 82
Why is the program called MATLAB? 82
What is a matrix? 82
Do I need to know any special types of matrices? 83
What is a vector? 83
How do I define a vector or a matrix in MATLAB? 84
What are some basic functions and commands for matrix manipulation? 86
How can I reference strings as vectors? 89
Multiple Choice Quiz 91
Exercises 93

Lesson 2.7 – How to Debug Code 94
What is an error? 94
What is a warning? 95
How can I solve the problem with my code? 95
Can I pause my program part of the way through? 98
What if I cannot find the exact place of the error? 100
Multiple Choice Quiz 101
Exercises 102

MODULE 3: PLOTTING

Lesson 3.1 – Plots and Figures 106
How can I visualize (plot) data in MATLAB? 106
How do I plot data pairs (points) in MATLAB? 106
How do I plot a function in MATLAB? 107
What is the difference between a figure and a plot? 109
How can I enter nonlinear functions for plotting? 109
What are some possible errors with plotting? 110
How do I show multiple data sets on the same plot? 110
What are some other types of plots that MATLAB can generate? 112
How do I plot on more than one figure in the same m-file? 113
What is the close all command? 114
Multiple Choice Quiz 115
Exercises 117

Lesson 3.2 – Plot Formatting 119
How can my MATLAB graph look nicer? 119
What are some terms I should know for plots? 119
How can I change the color and style of lines and markers on a plot? 120

How can I make the function and points on the graph look nicer? 121
How can I put a title and axis labels on my plot? 123
How can I add a legend to my plot? 123
How can I add a grid to my graph? 124
How can I add special characters in my axis labels and title? 125
How can I change axis limits and tick labels? 126
Multiple Choice Quiz 129
Exercises 130

Lesson 3.3 – Advanced Plotting 132
Does MATLAB have more plotting capabilities? 132
How can I create a bar graph? 132
How can I create a 3D line plot? 134
How can I create a 3D surface plot? 135
How can I create a polar plot? 138
Multiple Choice Quiz 141
Exercises 142

MODULE 4: MATH AND DATA ANALYSIS

Lesson 4.1 – Basic Algebra, Logarithms, and Trigonometry 144
What kind of mathematical functions and operations are available in MATLAB? 144
How do I use logarithmic functions in MATLAB? 144
What about a logarithm that is not natural? 146
How can MATLAB evaluate trigonometric functions? 147
Multiple Choice Quiz 151
Exercises 152

Lesson 4.2 – Symbolic Variables 155
What is a symbolic variable? 155
What is a MATLAB toolbox? 155
How do I use symbolic variables? 155
How do I clear specific variables? 157
How can I convert from syms data type to other data types? 157
Can I replace a symbolic variable with a value? 158
How can I change the output format of syms? 159
Multiple Choice Quiz 161
Exercises 163

Lesson 4.3 – Solution of Linear and Nonlinear Equations 165
How do I solve for roots of a linear equation? 165
What is a nonlinear equation? 166
How can I use MATLAB to solve nonlinear equations? 167
Is there a faster way to work with polynomial equations in MATLAB? 171
Can I plot with symbolic variables? 172
Multiple Choice Quiz 174

Exercises 175

Lesson 4.4 – Differential Calculus 179
What is a derivative? 179
How do I take the derivative of a function in MATLAB? 180
Where are derivatives used in engineering? 183
How do I find the derivative of a discrete function in MATLAB? 184
Multiple Choice Quiz 187
Exercises 189

Lesson 4.5 – Integral Calculus 191
What is integration? 191
How does MATLAB conduct symbolic integration? 192
Can MATLAB do numerical integration? 194
Multiple Choice Quiz 197
Exercises 199

Lesson 4.6 – Linear Algebra 202
What is linear algebra? 202
How do I add and subtract matrices? 202
Can I use math functions like sin() on matrices? 203
How do I perform matrix multiplication? 205
What is the difference between matrix and array operations? 205
How do I take the inverse of a matrix? 208
Can MATLAB do advanced matrix and vector operations? 210
How can I solve systems of equations with MATLAB? 213
Multiple Choice Quiz 219
Exercises 220

Lesson 4.7 – Curve Fitting 222
What is curve fitting? 222
What is interpolation? 222
How can I interpolate data in MATLAB? 222
What is spline interpolation? 225
How do I conduct spline interpolation? 226
What is regression? 228
How do I do regression in MATLAB? 229
Multiple Choice Quiz 233
Exercises 235

Lesson 4.8 – Curve Fitting – Plotting 238
How can I plot the results of curve fitting? 238
What are some common mistakes when plotting curve fitting results? 242
Multiple Choice Quiz 244
Exercises 246

Lesson 4.9 – Ordinary Differential Equations 249
What is a differential equation? 249
How do I set up and solve a differential equation? 250
How do I solve a higher order ODE? 251

What are the limitations of using the dsolve() function? 253
Multiple Choice Quiz 254
Exercises 256

MODULE 5: CONDITIONAL STATEMENTS

Lesson 5.1 – Conditions and Boolean Logic 260
What are conditions? 260
What is Boolean logic? 262
Can different data types be identified in MATLAB? 264
How can I round numbers in MATLAB? 265
Multiple Choice Quiz 268
Exercises 269

Lesson 5.2 – Conditional Statements: if and if-else 270
What is a conditional statement? 270
What is the if statement? 270
What is the if-else statement? 273
Can I use multiple conditions in a single expression? 275
A Note on Writing Good Conditions 278
Multiple Choice Quiz 279
Exercises 281

Lesson 5.3 – Conditional Statements: if-elseif 284
What is the if-elseif statement? 284
Independent vs. Dependent Cases 286
What is the if-elseif-else statement? 288
What is the difference between the else and elseif conditional clauses? 290
Multiple Choice Quiz 291
Exercises 293

MODULE 6: PROGRAM DESIGN AND COMMUNICATION

Lesson 6.1 – Flowcharts 298
What is a flowchart? 298
Multiple Choice Quiz 304
Exercises 306

Lesson 6.2 – Pseudocode 308
What is a pseudocode? 308
How are pseudocodes used? 308
How can I convert a pseudocode for a problem into a program? 310
Multiple Choice Quiz 312
Exercises 313

Lesson 6.3 – Writing Better Code 314
How can I improve my code for computational efficiency? 314

How does hardcoding impact a program? 317
What are some tips for good comments and spacing? 317
Why does proper code indenting matter? 318
What are some tips for choosing inputs and outputs? 318
What are some tips for thinking ahead in when designing my program? 318

MODULE 7: FUNCTIONS

Lesson 7.1 – User-Defined Functions 320
What is a function? 320
What are the naming rules for functions in MATLAB? 320
How can I create functions in MATLAB? 321
Can I define functions in the program m-file? 325
Multiple Choice Quiz 331
Exercises 333

Lesson 7.2 – Function Design and Communication 338
How can I add a description for my function? 338
How can I define errors and warnings inside my function? 339
Multiple Choice Quiz 342
Exercises 343

MODULE 8: LOOPS

Lesson 8.1 – while Loops 346
What is a loop? 346
What is a while loop? 347
What comparisons can I use with a while loop? 353
Multiple Choice Quiz 357
Exercises 360

Lesson 8.2 – for Loops 363
What is a for loop? 363
How can I reference vectors inside of a loop? 367
Do I have to use the loop counter variable in the body of the loop? 370
When do I use a for loop vs. a while loop? 372
Multiple Choice Quiz 375
Exercises 378

Lesson 8.3 – break and continue Commands 381
What are the break and continue commands? 381
How does the break command work in MATLAB? 381
How does the continue command work in MATLAB? 383
Multiple Choice Quiz 386
Exercises 389

Lesson 8.4 – Nested Loops 390
What is a nested loop? 390
How do nested loops work? 391
How do loop mechanics apply to nested loops? 392
What is a "flag"? 393
When should I use programming flags? 394
How can I use break and continue in nested loops? 396
Multiple Choice Quiz 399
Exercises 402

Lesson 8.5 – Working with Matrices and Loops 403
How can I reference matrices in a loop? 403
How do I store values in a matrix using a loop? 406
How can I access specific areas of a matrix? 409
What is vectorization? 415
How can I vectorize matrix operations in MATLAB? 416
What are some tips I can use for vectorization? 420
Multiple Choice Quiz 421
Exercises 424

Lesson 8.6 – Applied Loops 429
Why is this lesson important? 429
How can I sort an array? 429
How can I find the sum of a vector? 433
How can I plot different variations of a function using loops? 434
Multiple Choice Quiz 437
Exercises 440

MODULE 9: READING FROM AND WRITING TO FILES

Lesson 9.1 – Reading from Files 446
Why read data from a file? 446
How do I read numeric-only data from files? 446
What is a delimiter? 448
How can I read numeric and character data from files? 448
Multiple Choice Quiz 453
Exercises 454

Lesson 9.2 – Writing to Files 457
How can I write numeric data to files with MATLAB? 457
How can I write non-numeric data to files with MATLAB? 458
Multiple Choice Quiz 462
Exercises 463

Lesson 9.3 – Navigating Directories with MATLAB 465
How do I set the current working directory for MATLAB? 465
How can I loop through the contents of a directory? 467
How can I create a new folder (directory) with MATLAB? 470

Multiple Choice Quiz 472
Exercises 473

APPENDICES

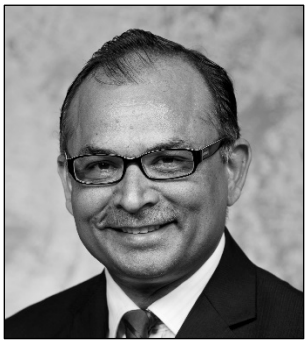
Appendix A – Matrix Algebra Primer 478
What is a matrix? 478
What are the special types of matrices? 479
Vector 479
Row Vector 479
Column Vector 479
Square Matrix 480
Trace of a Matrix 480
Upper Triangular Matrix 480
Lower Triangular Matrix 481
Diagonal Matrix 481
Identity Matrix 482
Zero Matrix 482
Tridiagonal Matrix 482
Diagonally Dominant Matrix 483
When are two matrices considered to be equal? 484
How do you add two matrices? 484
How do you subtract two matrices? 486
How do I multiply two matrices? 487
What is a scalar product of a constant and a matrix? 489
What is a linear combination of matrices? 490
What are some of the rules of binary matrix operations? 490
Commutative law of addition 490
Associate law of addition 491
Associate law of multiplication 491
Distributive law 491
Is [A][B]=[B][A]? 492
Transpose of a matrix 492
Symmetric matrix 493
Matrix algebra is used for solving a system of equations. 494
Can you divide two matrices? 496
Can I use the concept of the inverse of a matrix to find the solution of a set of equations [A] [X] = [C]? 497
How do I find the inverse of a matrix? 497

Appendix B – Mini Projects 501

Appendix C – Plot Animation 517

ABOUT THE AUTHORS

AUTAR KAW



Autar Kaw is a Professor of Mechanical Engineering at the University of South Florida, Tampa. He is a recipient of the 2012 U.S. Professor of the Year Award (doctoral and research universities) from the Council for Advancement and Support of Education and the Carnegie Foundation for Advancement of Teaching.

Professor Kaw obtained his B.E. (Hons.) degree in Mechanical Engineering from Birla Institute of Technology and Science, India in 1981. He received his Ph.D. degree in 1987 and M.S. degree in 1984, both in Engineering Mechanics from Clemson University, SC. He joined the faculty of the University of South Florida, Tampa in 1987. He has also been a Maintenance Engineer (1982) for Ford-Escorts Tractors, India, and a Summer Faculty Fellow (1992) and Visiting Scientist (1991) at the Wright Patterson Air Force Base.

Professor Kaw's current main scholarly interests are in engineering education research, adaptive learning, blended classroom, flipped learning, open courseware development, and the state and future of higher education. His research has been funded by National Science Foundation, Air Force Office of Scientific Research, Florida Department of Transportation, Research and Development Laboratories, Systran Co, Wright Patterson Air Force Base, and Montgomery Tank Lines.

He is a Fellow of the American Society of Mechanical Engineers (ASME) and a member of the American Society of Engineering Education (ASEE). He has authored textbooks on *Mechanics of Composite Materials*, and *Introduction to Matrix Algebra*, and co-authored a book on *Numerical Methods with Applications*. He is also a contributor to the MaterialsnetBase, an online library of material science texts, references, and handbooks.

Funded by the National Science Foundation, under Professor Kaw's leadership, he and his colleagues from around the nation have developed, implemented, refined, and assessed online resources for an open courseware (OCW) in Numerical Methods. This courseware annually receives 1,000,000+ page views, 2,000,000+ views of the YouTube lectures, and 90,000+ visitors to the "numerical methods guy" blog.

He has written more than 100 refereed 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 Chronicle of Higher Education, U.S. Congressional Record, Florida Senate Resolution, ASEE Prism, and Voice of America.

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 Tampa, FL 33620-5350

ACKNOWLEDGMENTS

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 THE THIRD 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.

- A companion website (MechPlus.org) has been constructed for easy reference and review. This is not a replacement for the book, but it allows students to quickly jump around to different lessons and see programming examples and explanations on the go. For specifics on the textbook, like where to purchase and have access to other resources, head to programming.autarkaw.com.

- 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.

Lesson 2.1
Variables and Naming Rules

After reading this lesson, you should be able to:

- *define a mathematical variable,*
- *define a programming variable,*
- *determine legal and illegal variable names,*
- *know the benefits of good practices for variable naming,*
- *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.

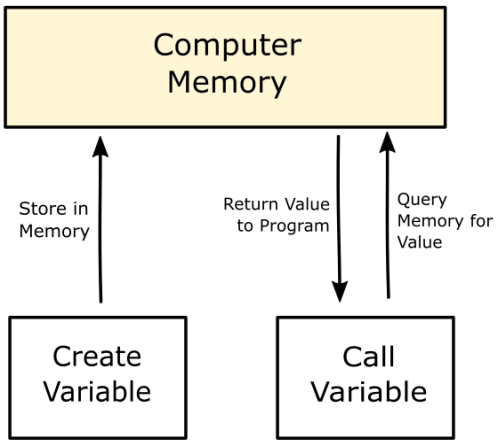


Figure 1: A simple visual representation of how variables are saved and called.

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

```
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
Lesson 2.1
Variables and Naming Rules

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

EXERCISES
Lesson 2.1
Variables and Naming Rules

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, $I = \frac{V}{R}$.

Lesson 3.2
Plot Formatting

After reading this lesson, you should be able to:

- *add axis labels to MATLAB plots,*
- *add a legend to MATLAB plots,*
- *add a title to MATLAB plots,*
- *add special characters to text in MATLAB plots,*
- *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.

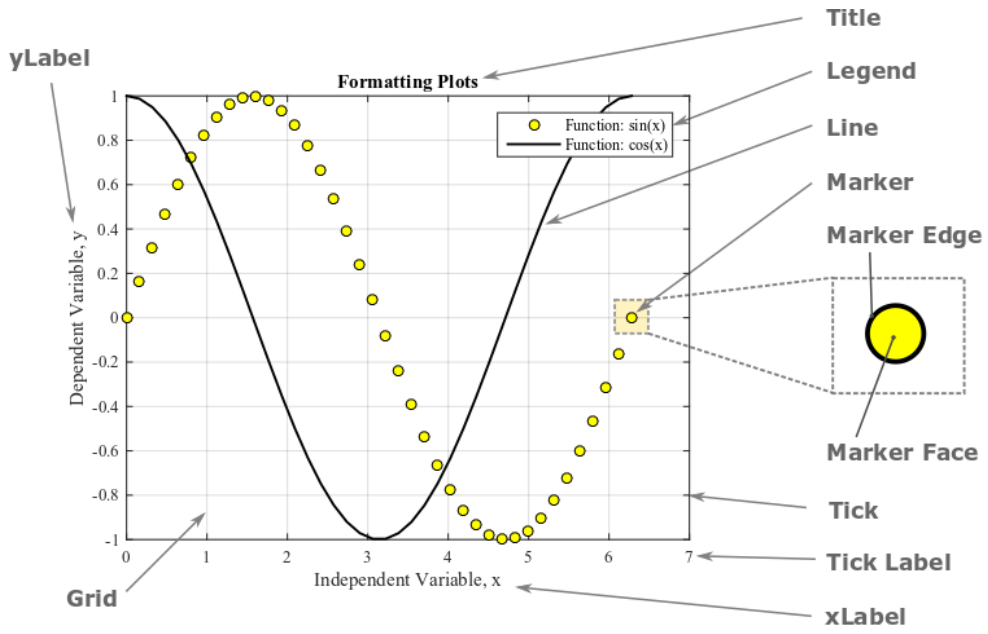


Figure 1: A MATLAB plot with common plot properties annotated.

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')`
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
```

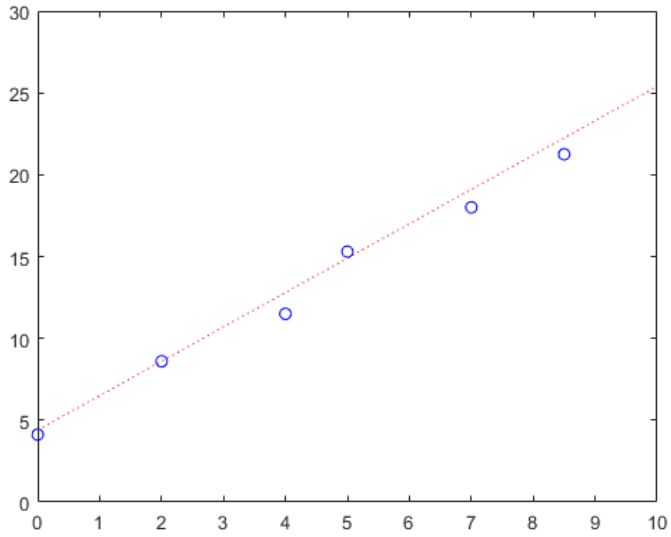


Figure 2: The figure output by the code in Example 1.

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
```

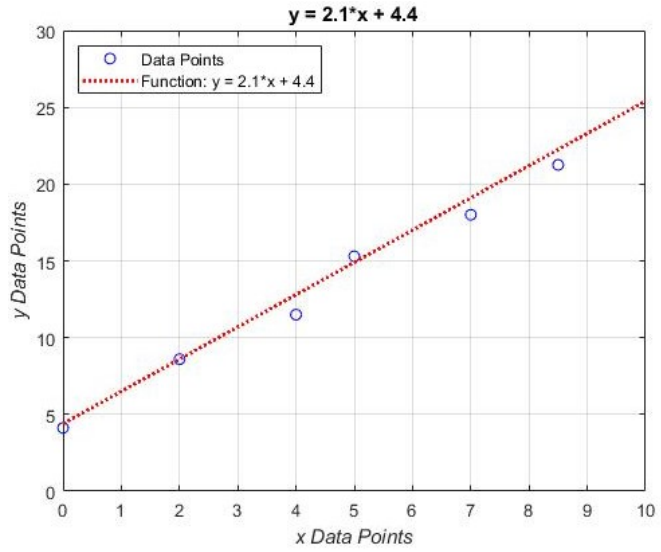


Figure 3: The figure output by the code in Example 2.

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')`
θ	`\theta`	`xlabel('\theta')`
π	`\pi`	`ylabel('\pi')`
σ	`\sigma`	`legend('\sigma')`
τ	`\tau`	`xlabel('\tau')`
±	`\pm`	`ylabel('\pm')`
÷	`\div`	`legend('\div')`

For example, to place, **`c = 2*π*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
```

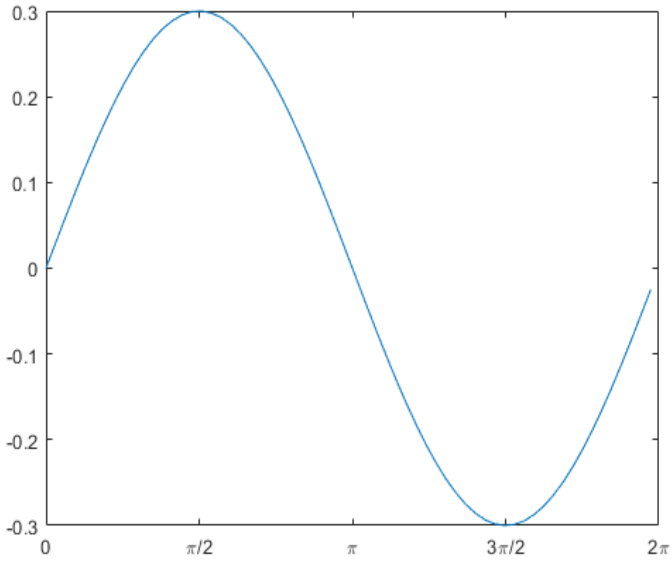


Figure 4: Plot with custom axis limits and axis labels on the horizontal axis (output for Example 3).

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
Lesson 3.2
Plot Formatting

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.

EXERCISES
Lesson 3.2
Plot Formatting

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

$$s(t)=\frac{3}{50}t^3 + \frac{7}{30}t^2 - 5t \text{ (ft).}$$

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:

$$\text{Exit pressure (bar)} = \begin{bmatrix}1.0 & 1.4 & 1.8 & 2.0 & 2.4 & 2.8 & 3.2\end{bmatrix}$$
$$h_{out}\left(\frac{\text{kJ}}{\text{kg}}\right) = \begin{bmatrix}278.76 & 286.96 & 295.45 & 304.50 & 313.49 & 332.60 & 342.21\end{bmatrix}$$

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: $F = \frac{T}{r}$. 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

Lesson 4.7
Curve Fitting

After reading this lesson, you should be able to:

- *conduct polynomial interpolation using MATLAB,*
- *conduct spline interpolation* using MATLAB*,*
- *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, $(x_0, y_0), (x_1, y_1), \ldots\ldots, (x_{n-1}, y_{n-1}), (x_n, y_n)$. 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()`.

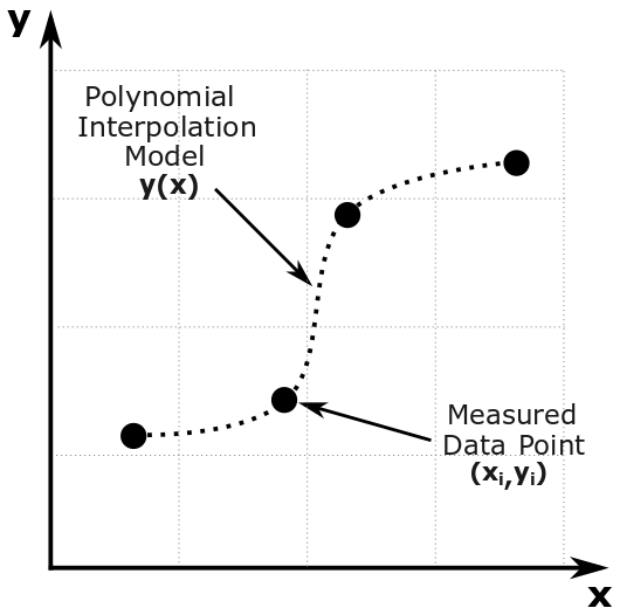


Figure 1: Interpolation of discrete data.

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 must be exactly one less than the total number of data pairs. So for given data $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$, the polynomial obtained would be of the form $y = a_1 x^n + a_2 x^{n-1} + \ldots + a_n$.

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 $[a_1, a_2, \ldots, a_n]$. 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.

x	1.0	4.0	8.0
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.

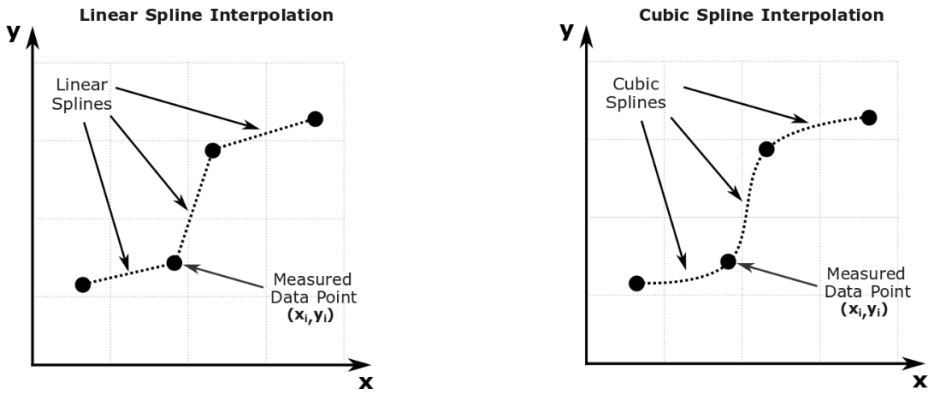


Figure 2: Spline interpolation of discrete data.

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!

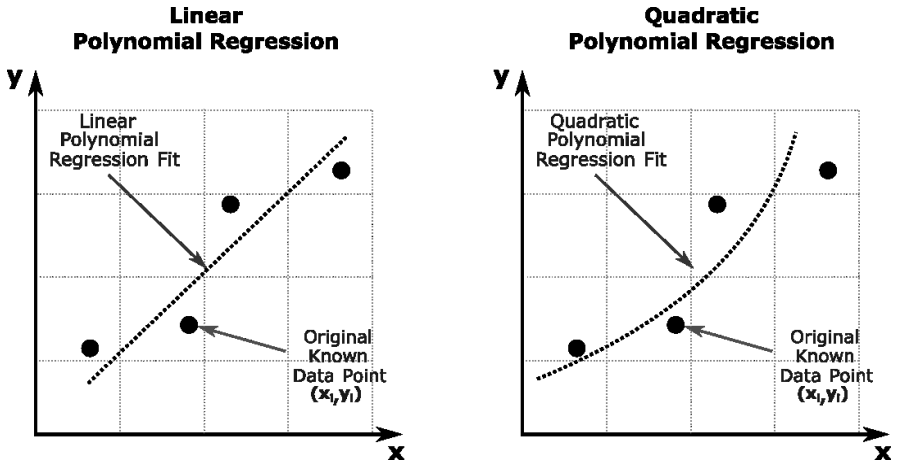


Figure 3: Regression of *n* data points to best fit a given order polynomial.

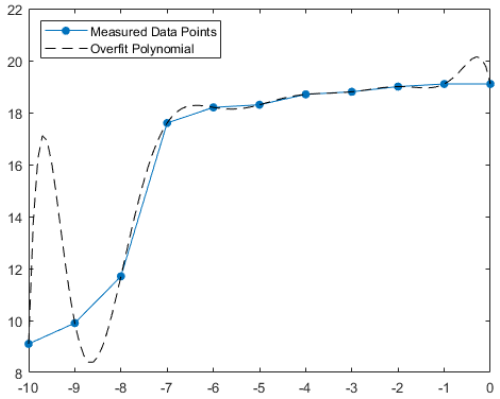


Figure 4: An example of overfitting on position and time data from an accelerating car (code not shown).

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 $(x_1, y_1),\ldots,(x_n, y_n)$, the polynomial obtained would be of the form $y = a_1x^m + a_2x^{m-1} + \ldots + a_m$, $1 \le m \le n-2$. 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 $[a_1, a_2,\ldots, a_m]$. 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()`.

Table C: Data pairs to be used for Example 1.

x	340	280	200	120	40	40	80
y	2.45	3.33	4.30	5.09	5.72	6.24	6.47

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

```
%----------------------------- 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)`
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
Lesson 4.7
Curve Fitting

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
xData = 1:1:5;
yData = [1 4 9 16 25];
coef  = polyfit(xData,yData,4);
syms x
model = poly2sym(coef,x)
```

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()`

EXERCISES
Lesson 4.7
Curve Fitting

1. Given are (x, y) data pairs in Table A.

Table A: Data pairs for Exercise 1.

x	1.4	2.3	5.0	7.5
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.

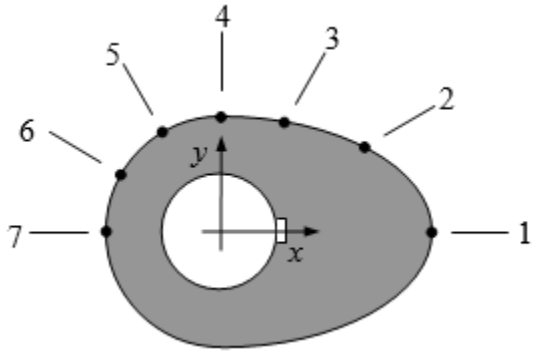


Figure A: Schematic of cam profile

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
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 (°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}

Find the coefficient of thermal expansion when the temperature is −150°F using
a. linear polynomial regression,
b. quadratic polynomial regression, and
c. cubic spline interpolation.

Lesson 5.1
Conditions and Boolean Logic

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 <, ≤, >, 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
Lesson 5.1
Conditions and Boolean Logic

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`

EXERCISES
Lesson 5.1
Conditions and Boolean Logic

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.

Lesson 8.1
while Loops

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.

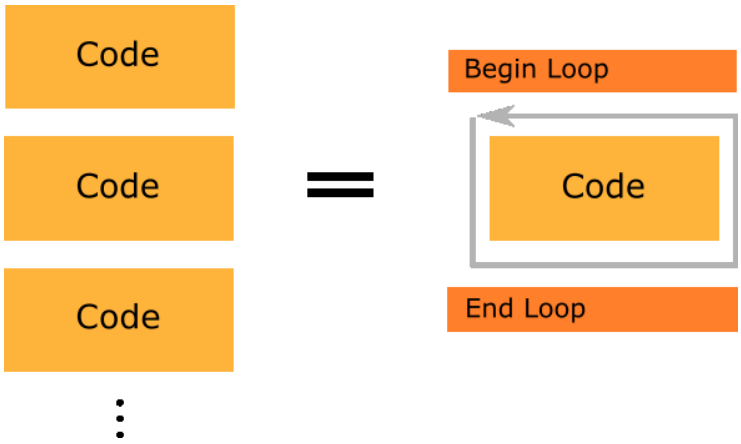


Figure 1: The fundamental structure of loops is to run the same block of code many times.

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.

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')

%----------------------------- 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')
i = startNum;
while i <= endNum
    num = i^2;  %Square each successive 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)
    i = i + increment;  %Increment the value of i
end
```

Command Window Output Example 1

```
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 `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

MATLAB Code example2.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To write a while loop that counts to four.
fprintf('To write a while loop that counts to four.\n\n')

%------------------------- SOLUTION/OUTPUTS ------------------------
fprintf('OUTPUTS\n')
i = 0;    %Initializing the loop variable (iterator)

%Beginning Loop: Defining while loop with a condition
while i < 4
    i = i + 1;                          %Loop Body: Adding to iterator
    fprintf('Loop iteration #%g.\n',i);  %Prints the current loop iterator
%Ending Loop: Defining the end of the while loop
end
```

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`).

Command Window Output Example 2

```
PURPOSE
To write a while loop that counts to four.

OUTPUTS
Loop iteration #1.
Loop iteration #2.
Loop iteration #3.
Loop iteration #4.
```

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 $x = -10, -9 \ldots 9, 10$.

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*

MATLAB Code example3.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To conditionally display the result of a mathematical function
fprintf('To conditionally display the result of a mathematical\n')
fprintf('function.\n\n')

%------------------------- SOLUTION/OUTPUTS ------------------------
fprintf('OUTPUTS\n')
x = -10;              %Starting point of x
while x <= 10
    y = x^2 - 49;
    if y > 0
        fprintf('y(%g) = %g\n',x,y)
    end
    x = x + 1;        %Adding 1 to the count
end
```

Command Window Output Example 3

```
PURPOSE
To conditionally display the result of a mathematical
function.

OUTPUTS
y(-10) = 51
y(-9) = 32
y(-8) = 15
y(8) = 15
y(9) = 32
y(10) = 51
```

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.

MATLAB Code example4.m

```
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 to conduct
fprintf('The starting value of x is %g.\n',x)
fprintf('x will be updated %g times.\n\n',numReps)

%----------------------------- SOLUTION -----------------------------
i = 1;                  %Starting point for loop counter
while i <= numReps
    x = 0.5*(x+(9/x));  %Finding the updated value of x
    i = i + 1;          %Going to the next step
end

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The updated value of x is %g.\n',x)
```

Command Window Output Example 4

```
PURPOSE
To implement a recursive mathematical formula.

INPUTS
The starting value of x is 64.
x will be updated 10 times.

OUTPUTS
The updated value of x is 3.
>>
```

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.

Example 5

The value of the exponential function, e^x, can be found by the following series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots.$$

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

$$\text{Absolute Relative Approximate Error} = \frac{|\text{Previous Approximation - Present Approximation}|}{|\text{Present Approximation}|} \times 100\ \%$$

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

$$e^x = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots.$$

and hence in the compact mathematical form as

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

MATLAB Code example5.m

```
clc
clear

%----------------------------- PURPOSE -----------------------------
fprintf('PURPOSE\n')
%To approximate a value of the exponential function
fprintf('To approximate a value of the exponential function.\n\n')

%----------------------------- INPUTS -----------------------------
fprintf('INPUTS\n')
x         = 0.75;
tolerance = 0.1;
fprintf('Evaluate the exponential function at %g.\n',x)
fprintf('The desired relative error tolerance is %g%%.\n\n',tolerance)

%----------------------------- SOLUTION -----------------------------
term = 1;               %Defining term counter for the series
exp1 = 0;               %Starting value of the series
prevExp = 0;            %Defining variable to store previous approximation
ARAE = tolerance + 1;   %Initializing value for ARAE to be greater than
                        %    tolerance
maxTerms = 10;          %Defining the max number of terms that will be used

while ARAE > tolerance && term <= maxTerms
    prevExp = exp1;     %Storing the previous value before it is updated
                        %   on the next line
    exp1 = exp1 + (x^(term-1))/factorial(term-1);  %Updating approximation
    ARAE = abs((prevExp-exp1)/exp1)*100;           %Calculating error
    term = term + 1;                               %Iterating term counter
end

%----------------------------- OUTPUTS -----------------------------
fprintf('OUTPUTS\n')
fprintf('The approximate value is exp(%g) = %.4f using\n',x,exp1)
fprintf('   %g terms of the series.\n',term)
fprintf('The abs. rel. approximate error at the end is %.4f%%.\n',ARAE)
```

Command Window Output Example 5

```
PURPOSE
To approximate a value of the exponential function.

INPUTS
Evaluate the exponential function at 0.75.
The desired relative error tolerance is 0.1%.

OUTPUTS
The approximate value is exp(0.75) = 2.1167 using
   7 terms of the series.
The abs. rel. approximate error at the end is 0.0934%.
>>
```

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 of New Syntax and Programming Tools

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
Lesson 8.1
while Loops

1. The `while` loop is
a) a definite loop.
b) not a loop.
c) an indefinite loop.
d) an infinite loop.

2. The Command Window output of the following program is

```
clc
clear
i = 0;
while i <= 3
      abc = 1;
      i = i + 1;
end
abc
```

a) `abc = 1`
b) `abc = 2`
c) `abc = 3`
d) `abc = 4`

3. The Command Window output of the following program is

```
clc
clear
i = 0;
while i <= 3
      i = i + 1;
      abc = i;
end
abc
```

a) `abc = 1`
b) `abc = 3`
c) `abc = 4`
d) `abc = 5`

4. The maximum number of logical comparisons that can be made in the conditional expression in the definition of each `while` loop is

a) 0
b) 1
c) 2
d) as many as needed.

5. The Command Window output of the following program is

```
clc
clear
i = 0;
while i <= 4
    j = i*3;
    i = i + 1;
end
j
```

a) `j = 0`
b) `j = 5`
c) `j = 12`
d) `j = 15`

6. The Command Window output of the following program is

```
clc
clear
i = 0;
while i < =4
      i = i + 1;
      j = i*3;
end
j
```

a) `j = 0`
b) `j = 5`
c) `j = 12`
d) `j = 15`

7. The Command Window output of the following program is

```
clc
clear
a = 2;
i = 2;
while i <= 4
      i = i + 1;
      abc = a*i;
end
abc
```

a) `abc = 1`
b) `abc = 5`
c) `abc = 10`
d) `abc = 15`

EXERCISES
Lesson 8.1
while Loops

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 = [2 5 8 -4 7 0 -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

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots.$$

The absolute percentage relative approximate error, $|\varepsilon_a|$ is defined as

$$|\varepsilon_a| = \left| \frac{\text{Present Approximat ion} - \text{Previous Approximat ion}}{\text{Present Approximat ion}} \right| \times 100.0$$

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:

$$S = a + ar + ar^2 + \ldots + ar^n$$

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`.