

DATA AND PROGRAMMING- AN OVERVIEW



Julio Terán
North Carolina State University



Data and programming- an overview

Julio Terán

North Carolina State University

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (<https://LibreTexts.org>) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by [NICE CXone Expert](#) and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 05/15/2026

TABLE OF CONTENTS

Licensing

1: What Are Data and Data Science?

- 1.1: Introduction
- 1.2: What Is Data Science?
- 1.3: Data Science in Practice
- 1.4: Data and Datasets
- 1.5: Using Technology for Data Science
- 1.6: Correlation and Linear Regression Analysis
- 1.7: Key Terms
- 1.8: Chapter Review
- 1.9: References

2: Units, Dimensions, and Conversions

3: Introduction to programming and logic

- 3.1: Thinking Algorithmically
- 3.2: Relational and Logical Operators
- 3.3: Documentation and such
- 3.4: Programming in any language
- 3.5: Mathematical Models

4: MATLAB

- 4.1: Introduction to MATLAB
- 4.2: MATLAB Scripting and Arrays
 - 4.2.1: MATLAB Decisions
- 4.3: MATLAB Loops
- 4.4: MATLAB Plotting
- 4.5: User Defined Functions

Index

Glossary

Detailed Licensing

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

CHAPTER OVERVIEW

1: What Are Data and Data Science?

- [1.1: Introduction](#)
- [1.2: What Is Data Science?](#)
- [1.3: Data Science in Practice](#)
- [1.4: Data and Datasets](#)
- [1.5: Using Technology for Data Science](#)
- [1.6: Correlation and Linear Regression Analysis](#)
- [1.7: Key Terms](#)
- [1.8: Chapter Review](#)
- [1.9: References](#)

This page titled [1: What Are Data and Data Science?](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.1: Introduction



Figure 1.1 Petroglyphs are one of the earliest types of data generated by humanity, providing vital information about the daily life of the people who created them. (credit: modification of work “Indian petroglyphs (~100 B.C. to ~1540 A.D.) (Newspaper Rock, southeastern Utah, USA) 24” by James St. John/Flickr, CC BY 2.0)

Many of us use the terms “data” and “data science,” but not necessarily with a lot of precision. This chapter will define data science terminology and apply the terms in multiple fields. The chapter will also briefly introduce the types of technology (such as statistical software, spreadsheets, and programming languages) that data scientists use to perform their work and will then take a deeper dive into the use of Python for data analysis. The chapter should help you build a technical foundation so that you can practice the more advanced data science concepts covered in future chapters.

This page titled [1.1: Introduction](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.2: What Is Data Science?

Learning Objectives

By the end of this section, you should be able to:

- 1.1.1 Describe the goals of data science.
- 1.1.2 Explain the data science cycle and goals of each step in the cycle.
- 1.1.3 Explain the role of data management in the data science process.

Data science is a field of study that investigates how to collect, manage, and analyze data of all types in order to retrieve meaningful information. Although we will describe data in more detail in Data and Datasets, you can consider *data* to be any pieces of evidence or observations that can be analyzed to provide some insights.

In its earliest days, the work of data science was spread across multiple disciplines, including statistics, mathematics, computer science, and social science. It was commonly believed that the job of data collection, management, and analysis would be carried out by different types of experts, with each job independent of one another. To be more specific, *data collection* was considered to be the province of so-called domain experts (e.g., doctors for medical data, psychologists for psychological data, business analysts for sales, logistic, and marketing data, etc.) as they had a full context of the data; *data management* was for computer scientists/engineers as they knew how to store and process data in computing systems (e.g., a single computer, a server, a data warehouse); and *data analysis* was for statisticians and mathematicians as they knew how to derive some meaningful insights from data. Technological advancement brought about the proliferation of data, muddying the boundaries between these jobs, as shown in Figure 1.2. Now, it is expected that a data scientist or data science team will have some expertise in all three domains.

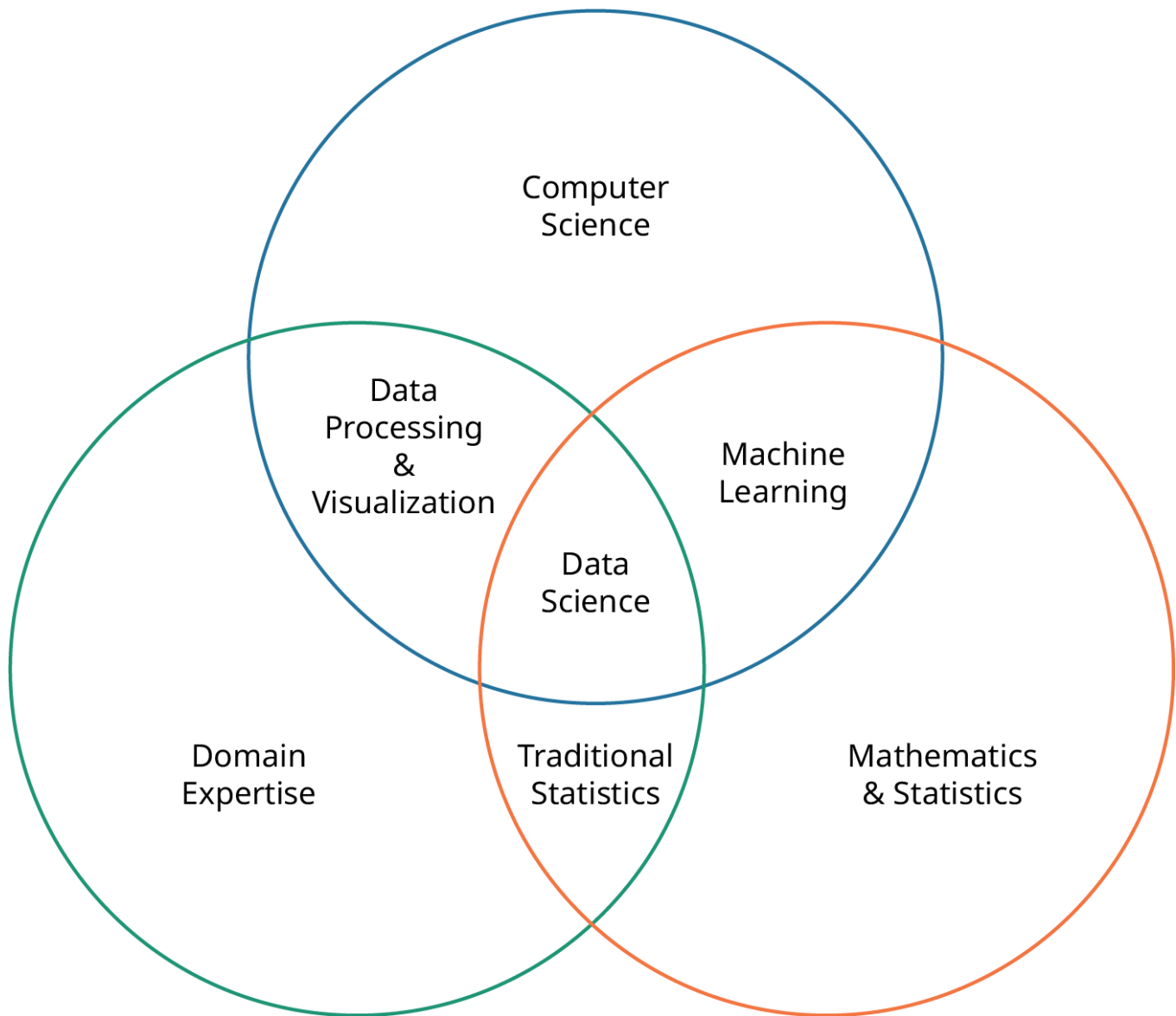


Figure 1.2 The Field of Data Science

One good example of this is the development of personal cell phones. In the past, households typically had only one landline telephone, and the only data that was generated with the telephone was the list of phone numbers called by the household members. Today the majority of consumers own a smartphone, which contains a tremendous amount of data: photos, social media contacts, videos, locations (usually), and perhaps health data (with the consumers' consent), among many other things.

Is the data from a smartphone solely collected by domain experts who are specialized in photos, videos, and such? Probably not. They are automatically logged and collected by the smartphone system itself, which is designed by computer scientists/engineers. For a health care scientist to collect data from many individuals in the “traditional” way, bringing patients into a laboratory and taking vital signs regularly over a period of time takes a lot of time and effort. A smartphone application is a more efficient and productive method, from a data collection perspective.

Data science tasks are often described as a process, and this section provides an overview for each step of that process.

The Data Science Cycle

Data science tasks follow a process, called the data science cycle, which includes problem definition, then data collection, preparation, analysis, and reporting, as illustrated in Figure 1.3. See this [animation about data science](#), which describes the data science cycle in much the same way.



Figure 1.3 The Data Science Cycle

Although data collection and preparation may sound like simple tasks compared to the more important work of analysis, they are actually the most time- and effort-consuming steps in the data science cycle. According to a survey conducted by Anaconda (2020), data scientists spend about half of the entire process in data collection and cleaning, while data analysis and communication take about a quarter to a third of the time each, depending on the job.

Problem Definition, Data Collection, and Data Preparation

The first step in the data science cycle is a precise definition of the problem statement to establish clear objectives for the goal and scope of the data analysis project. Once the problem has been well defined, the data must be generated and collected. Data collection is the systematic process of gathering information on variables of interest. Data is often collected purposefully by domain experts to find answers to predefined problems. One example is data on customer responses to a product satisfaction survey. These survey questions will likely be crafted by the product sales and marketing representatives, who likely have a specific plan for how they wish to use the response data once it is collected.

Not all data is generated this purposefully, though. A lot of data around our daily life is simply a by-product of our activity. These by-products are kept as data because they could be used by others to find some helpful insights later. One example is our web search histories. We use a web search engine like Google to search for information about our interests, and such activity leaves a history of the search text we used on the Google server. Google employees can utilize the records of numerous Google users in order to analyze common search patterns, present accurate search results, and potentially, to display relatable advertisements back to the searchers.

Often the collected data is not in an optimal form for analysis. It needs to be processed somehow so that it can be analyzed, in the phase called data preparation or *data processing*. Suppose you work for Google and want to know what kind of food people around the globe search about the most during the night. You have users' search history from around the globe, but you probably cannot use the search history data as is. The search keywords will probably be in different languages, and users live all around the Earth, so nighttime will vary by each user's time zone. Even then, some search keywords would have some typos, simply not make sense, or even remain blank if the Google server somehow failed to store that specific search history record. Note that all these scenarios are possible, and therefore data preparation should address these issues so that the actual analysis can draw more accurate results. There are many different ways to manage these issues, which we will discuss more fully in Collecting and Preparing Data.

Data Analysis

Once the data is collected and prepared, it must be analyzed in order to discover meaningful insights, a process called data analysis. There are a variety of data analysis methods to choose from, ranging from simple ones like checking minimum and maximum values, to more advanced ones such as modelling a dependent variable. Most of the time data scientists start with simple methods and then move into more advanced ones, based on what they want to investigate further. Descriptive Statistics: Statistical Measurements and Probability Distributions and Inferential Statistics and Regression Analysis discuss when and how to use different analysis methods. Time Series and Forecasting and Decision-Making Using Machine Learning Basics discuss forecasting and decision-making.

Data Reporting

Data reporting involves the presentation of data in a way that will best convey the information learned from data analysis. The importance of data reporting cannot be overemphasized. Without it, data scientists cannot effectively communicate to their audience the insights they discovered in the data. Data scientists work with domain experts from different fields, and it is their responsibility to communicate the results of their analysis in a way that those domain experts can understand. Data visualization is a graphical way of presenting and reporting data to point out the patterns, trends, and hidden insights; it involves the use of visual

elements such as charts, graphs, and maps to present data in a way that is easy to comprehend and analyze. The goal of data visualization is to communicate information effectively and facilitate better decision-making. Data visualization and basic statistical graphing, including how to create graphical presentations of data using Python, are explored in depth in [Visualizing Data](#). Further details on reporting results are discussed in [Reporting Results](#).

Data Management

In the early days of data analysis (when generated data was mostly structured and not quite so “big”), it was possible to keep data in local storage (e.g., on a single computer or a portable hard drive). With this setup, data processing and analysis was all done locally as well.

When so much more data began to be collected—much of it unstructured as well as structured—cloud-based management systems were developed to store all the data on a designated server, outside a local computer. At the same time, data scientists began to see that most of their time was being spent on data processing rather than analysis itself. To address this concern, modern data management systems not only store the data itself but also perform some basic processing on a cloud. These systems, referred to as data warehousing, store and manage large volumes of data from various sources in a central location, enabling efficient retrieval and analysis for business intelligence and decision-making. (Data warehousing is covered in more detail in [Handling Large Datasets](#).)

Today, enterprises simply subscribe to a cloud-warehouse service such as Amazon RedShift (which runs on the Amazon Web Services) or Google BigQuery (which runs on the Google Cloud) instead of buying physical storage and configuring data management/processing systems on their own. These services ensure the data is safely stored and processed on the cloud, all without spending money on purchasing/maintaining physical storage.

This page titled [1.2: What Is Data Science?](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.3: Data Science in Practice

Learning Objectives

By the end of this section, you should be able to:

- 1.2.1 Explain the interdisciplinary nature of data science in various fields.
- 1.2.2 Identify examples of data science applications in various fields.
- 1.2.3 Identify current issues and challenges in the field of data science

While data science has adopted techniques and theories from fields such as math, statistics, and computer science, its applications concern an expanding number of fields. In this section we introduce some examples of how data science is used in business and finance, public policy, health care and medicine, engineering and sciences, and sports and entertainment.

Data Science in Business

Data science plays a key role in many business operations. A variety of data related to customers, products, and sales can be collected and generated within a business. These include customer names and lists of products they have purchased as well as daily revenue. Business analytics investigate these data to launch new products and to maximize the business revenue/profit.

Retail giant Walmart is known for using business analytics to improve the company's bottom line. Walmart collects multiple *petabytes* (1 petabyte = 1,024 terabytes) of unstructured data every hour from millions of customers (commonly referred to as “big data”); as of 2024, Walmart's customer data included roughly 255 million weekly customer visits (Statista, 2024). Walmart uses this big data to investigate consumer patterns and adjust its inventories. Such analysis helps the company avoid overstocking or understocking and resulted in an estimated online sales increase of between 10% and 15%, translating to an extra \$1 billion in revenue (ProjectPro, 2015). One often-reported example includes the predictive technology Walmart used to prepare for Hurricane Frances in 2004. A week before the hurricane's arrival, staff were asked to look back at their data on sales during Hurricane Charley, which hit several weeks earlier, and then come up with some forecasts about product demand ahead of Frances (Hays, 2004). Among other insights, the executives discovered that strawberry Pop-Tart sales increased by about sevenfold during that time. As a result, in the days before Hurricane Frances, Walmart shipped extra supplies of strawberry Pop-Tarts to stores in the storm's path (Hays, 2004). The analysis also provided insights on how many checkout associates to assign at different times of the day, where to place popular products, and many other sales and product details. In addition, the company has launched social media analytics efforts to investigate hot keywords on social media and promptly make related products available (ProjectPro, 2015).

Amazon provides another good example. Ever since it launched its Prime membership service, Amazon has focused on how to minimize delivery time and cost. Like Walmart, it started by analyzing consumer patterns and was able to place products close to customers. To do so, Amazon first divided the United States into eight geographic regions and ensured that most items were warehoused and shipped within the same region; this allowed the company to reduce the shipping time and cost. As of 2023, more than 76% of orders were shipped from within the customer's region, and items in same-day shipping facilities could be made ready to put on a delivery truck in just 11 minutes (Herrington, 2023). Amazon also utilizes machine learning algorithms to predict the demand for items in each region and have the highest-demand items available in advance at the fulfillment center of the corresponding region. This predictive strategy has helped Amazon reduce the delivery time for each product and extend the item selections for two-day shipping (Herrington, 2023).

Data science is utilized extensively in finance as well. Detecting and managing fraudulent transactions is now done by automated machine learning algorithms (IABAC, 2023). Based on the customer data and the patterns of past fraudulent activities, an algorithm can determine whether a transaction is fraudulent in real time. Multiple tech companies, such as IBM and Amazon Web Services, offer their own fraud detection solutions to their corporate clients. (For more information, see this online resource on Fraud Detection through Data Analytics.)

Data Science in Engineering and Science

Various fields of engineering and science also benefit from data science. Internet of Things (IoT) is a good example of a new technology paradigm that has benefited from data science. Internet of Things (IoT) describes the network of multiple objects

interacting with each other through the Internet. Data science plays a crucial role in these interactions since behaviors of the objects in a network are often triggered by data collected by another object in the network. For example, a smart doorbell or camera allows us to see a live stream on our phone and alerts us to any unusual activity.

In addition, weather forecasting has always been a data-driven task. Weather analysts collect different measures of the weather such as temperature and humidity and then make their best estimate for the weather in the future. Data science has made weather forecasting more reliable by adopting more sophisticated prediction methods such as time-series analysis, artificial intelligence (AI), and machine learning (covered in Time Series and Forecasting, Decision-Making Using Machine Learning Basics, and Deep Learning and AI Basics). Such advancement in weather forecasting has also enabled engineers and scientists to predict some natural disasters such as flood or wildfire and has enabled precision farming, with which agricultural engineers can identify an optimal time window to plant, water, and harvest crops. For example, an agronomy consultant, Ag Automation, has partnered with Hitachi to offer a solution that both automates data collection and remotely monitors and controls irrigation for the best efficiency (Hitachi, 2023).

Exploring Further

Using AI for Irrigation Control

See this [Ag Automation video](#) demonstrating the use of data collection for controlling irrigation.

Data Science in Public Policy

Smart cities are among the most representative examples of the use of data science in public policy. Multiple cities around the world, including Masdar City in the United Arab Emirates and Songdo in South Korea, have installed thousands of data-collecting sensors used to optimize their energy consumption. The technology is not perfect, and smart cities may not yet live up to their full potential, but many corporations and companies are pursuing the goal of developing smart cities more broadly (Clewlow, 2024). The notion of a smart city has also been applied on a smaller scale, such as to a parking lot, a building, or a street of lights. For example, the city of San Diego installed thousands of sensors on the city streets to control the streetlights using data and smart technology. The sensors measure traffic, parking occupancy, humidity, and temperature and are able to turn on the lights only when necessary (Van Bocxlaer, 2020). New York City has adopted smart garbage bins that monitor the amount of garbage in a bin, allowing garbage collection companies to route their collection efforts more efficiently (Van Bocxlaer, 2020).

Exploring Further

Sensor Networks to Monitor Energy Consumption

See how [Songdo](#) monitors energy consumption and safety with sensor networks.

Data Science in Education

Data science also influences education. Traditional instruction, especially in higher education, has been provided in a one-size-fits-all form, such as many students listening to a single instructor's lecture in a classroom. This makes it difficult for an instructor to keep track of each individual student's learning progress. However, many educational platforms these days are online and can produce an enormous amount of student data, allowing instructors to investigate everyone's learning based on these collected data. For example, online learning management systems such as [Canvas](#) compile a grade book in one place, and online textbooks such as [zyBooks](#) collect students' mastery level on each topic through their performance on exercises. All these data can be used to capture each student's progress and offer personalized learning experiences such as intelligent tutoring systems or adaptive learning. [ALEKS](#), an online adaptive learning application, offers personalized material for each learner based on their past performance.

Data Science in Health Care and Medicine

The fields of health care and medicine also use data science. Often their goal is to offer more accurate diagnosis and treatment using predictive analytics—statistical techniques, algorithms, and machine learning that analyze historical data and make predictions about future events. Medical diagnosis and prescription practices have traditionally been based on a patient's verbal description of symptoms and a doctor's or a group of doctors' experience and intuition; this new movement allows health care

professionals to make decisions that are more data-driven. Data-driven decisions became more feasible thanks to all the personal data collected through personal gadgets—smartphones, smart watches, and smart bands. Such devices collect daily health/activity records, and this in turn helps health care professionals better capture each patient’s situation. All this work will enable patients to receive more accurate diagnoses along with more personalized treatment regimens in the long run.

The [Precision Medicine Initiative](#) is a long-term research endeavor carried out by the National Institutes of Health (NIH) and other research centers, with the goal of better understanding how a person's genetics, environment, and lifestyle can help determine the best approach to prevent or treat disease. The initiative aims to look at as much data as possible to care for a patient’s health more proactively. For example, the initiative includes genome sequencing to look for certain mutations that indicate a higher risk of cancer or other diseases.

Another application of data science in health focuses on lowering the cost of health care services. Using historical records of patients’ symptoms and prescription, a chatbot that is powered by artificial intelligence can provide automated health care advice. This will reduce the need for patients to see a pharmacist or doctor, which in turn improves health care accessibility for those who are in greater need.

Exploring Further

[Big Data In Health Care](#)

The 2015 launch of the National Institutes of Health Precision Medicine Initiative was documented in [One Woman’s Quest to Cure Her Fatal Brain Disease](#). “Promise of Precision Medicine” signaled a new approach to health care in the United States—one heavily reliant on big data.

Data Science in Sports and Entertainment

Data science is prevalent in the sports and the entertainment industry as well. Sports naturally produce much data—about the player, positions, teams, seasons, and so on. Therefore, just as there is the concept of business analytics, the analysis of such data in sports is called sports analytics. For example, the Oakland Athletics baseball team famously analyzed player recruitment for the 2002 season. The team’s management adapted a statistical approach referred to as sabermetrics to recruit and position players. With sabermetrics, the team was able to identify critical yet traditionally overlooked metrics such as on-base percentage and slugging percentage. The team, with its small budget compared to other teams, recruited a number of undervalued players with strong scores on these metrics, and in the process, they became one of the most exciting teams in baseball that year, breaking the American League record for 20 wins in a row. Does this story sound familiar? This story was so dramatic that Michael Lewis wrote a book about it, which was also made into a movie: *Moneyball*.

Exploring Further

[The Sabermetrics YouTube Channel](#)

Sabermetrics is so popular that there is a YouTube channel devoted to it: [Simple Sabermetrics](#), with baseball animations and tutorials explaining how data impacts the way today’s game is played behind the scenes.

In the entertainment industry, data science is commonly used to make data-driven, personalized suggestions that satisfy consumers known as recommendation systems. One example of a recommendation system is on video streaming services such as Netflix. Netflix Research considers subscribers’ watch histories, satisfaction with the content, and interaction records (e.g., search history). Their goal is to make perfect personalized recommendations despite some challenges, including the fact that subscribers themselves often not do not know what they want to see.

Exploring Further

[Careers in Data Science](#)

As you advance in your data science training, consider the many professional options in this evolving field. This helpful [graphic from edX](#) distinguishes data analyst vs. data science paths. This [Coursera article](#) lists typical skill sets by role. Current practitioner discussions are available on forums such as Reddit’s [r/data science](#).

Trends and Issues in Data Science

Technology has made it possible to collect abundant amounts of data, which has led to challenges in the processing and analyzing of that data. But technology comes to the rescue again! Data scientists now use machine learning to better understand the data, and artificial intelligence can make an automated, data-driven decision on a task. Decision-Making Using Machine Learning Basics and Deep Learning and AI Basics will cover the details of machine learning and artificial intelligence.

With these advances, many people have raised concerns about ethics and privacy. Who is allowed to collect these data, and who has access to them? None of us want someone else to use our personal data (e.g., contact information, health records, location, photos, web search history) without our consent or without knowing the risk of sharing our data. Machine learning algorithms and artificial intelligence are trained to make a decision based on the past data, and when the past data itself inherits some bias, the trained machine learning algorithms and artificial intelligence will make biased decisions as well. Thus, carefully attending to the process of collecting data and evaluating the bias of a trained results is critical. Ethics Throughout the Data Science Cycle will discuss these and other ethical concerns and privacy issues in more depth.

This page titled [1.3: Data Science in Practice](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.4: Data and Datasets

Learning Objectives

By the end of this section, you should be able to:

- 1.3.1 Define data and dataset.
- 1.3.2 Differentiate among the various data types used in data science.
- 1.3.3 Identify the type of data used in a dataset.
- 1.3.4 Discuss an item and attribute of a dataset.
- 1.3.5 Identify the different data formats and structures used in data science.

What Is Data Science? and Data Science in Practice introduced the many varieties of and uses for data science in today's world. Data science allows us to extract insights and knowledge from data, driving decision-making and innovation in business, health care, entertainment, and so on. As we've seen, the field has roots in math, statistics, and computer science, but it only began to emerge as its own distinct field in the early 2000s with the proliferation of digital data and advances in computing power and technology. It gained significant momentum and recognition around the mid to late 2000s with the rise of big data and the need for sophisticated techniques to analyze and derive insights from large and complex datasets. Its evolution since then has been rapid, and as we can see from the previous discussion, it is quickly becoming a cornerstone of many industries and domains.

Data, however, is not new! Humans have been collecting data and generating datasets from the beginning of time. This started in the Stone Age when people carved some shapes and pictures, called petroglyphs, on rock. The petroglyphs provide insights on how animals looked and how they carried out their daily life, which is valuable “data” for us. Ancient Egyptians invented a first form of paper—papyrus—in order to journal *their* data. Papyrus also made it easier to store data in bulk, such as listing inventories, noting financial transactions, and recording a story for future generations.

Data

“Data” is the plural of the Latin word “datum,” which translates as something that is given or used and is often used to mean a single piece of information or a single point of reference in a dataset. When you hear the word “data,” you may think of some sort of “numbers.” It is true that numbers are usually considered data, but there are many other forms of data all around us. Anything that we can analyze to compile information—high-level insights—is considered data.

Suppose you are debating whether to take a certain course next semester. What process do you go through in order to make your decision? First, you might check the course evaluations, as shown in Table 1.1.

Table 1.1 Course Evaluation Records

Semester	Instructor	Class Size	Rating
Fall 2020	A	100	Not recommended at all
Spring 2021	A	50	Highly recommended
Fall 2021	B	120	Not quite recommended
Spring 2022	B	40	Highly recommended
Fall 2022	A	110	Recommended
Spring 2023	B	50	Highly recommended

The evaluation record consists of four kinds of data, and they are grouped as columns of the table: Semester, Instructor, Class Size, and Rating. Within each column there are six different pieces of data, located at each row. For example, there are six pieces of text data under the Semester column: “Fall 2020,” “Spring 2021,” “Fall 2021,” “Spring 2022,” “Fall 2022,” and “Spring 2023.”

The course evaluation ratings themselves do not provide an idea on whether to take the course next semester. The ratings are just a *phrase* (e.g., “Highly recommended” or “Not quite recommended”) that *encodes* how recommended the course was in that

semester. You need to analyze them to come up with a decision!

Now let's think about how to derive the information from these ratings. You would probably look at all the data, including when in the semester the course was offered, who the instructor is, and class size. These records would allow you to derive information that would help you decide "whether or not to take the course next semester."

Example 1.1

Suppose you want to decide whether or not to put on a jacket today. You research the highest temperatures in the past five days and determine whether you needed a jacket on each day. In this scenario, what data are you using? And what information are you trying to derive?

Answer

Temperature readings and whether you needed a jacket on each of the past five days are two kinds of data you are referring to. Again, they do not indicate anything related to wearing a jacket today. They are simply five pairs of numbers (temperature) and yes/no (whether you needed a jacket) records, with each pair representing a day. Using these data, you are deriving information that you can analyze to help you decide whether to wear a jacket today.

Types of Data

The previous sections talked about how much our daily life is surrounded by data, how much our daily life itself produces new data, and how often we make data-driven decisions without even noticing it. You might have noticed that data comes in various types. Some data are quantitative, which means that they are measured and expressed using numbers. Quantitative data deals with quantities and amounts and is usually analyzed using statistical methods. Examples include numerical measurements like height, weight, temperature, heart rate, and sales figures. Qualitative data are non-numerical data that generally describe subjective attributes or characteristics and are analyzed using methods such as thematic analysis or content analysis. Examples include descriptions, observations, interviews, and open-ended survey responses (as we'll see in Survey Design and Implementation) that address unquantifiable details (e.g., photos, posts on Reddit). The data types often dictate methods for data analysis, so it is important to be able to identify a type of data. Thus, this section will take a further dive into types of data.

Let's revisit our previous example about deciding whether to take a certain course next semester. In that example, we referred to four pieces of data. They are encoded in different types such as numbers, words, and symbols.

1. The semester the course was offered—Fall 2020, Spring 2021, ..., Fall 2022, Spring 2023
2. The instructor—A and B
3. The class size—100, 50, 120, 40, 110, 50
4. The course rating—"Not recommended at all," ..., "Highly recommended"

There are two primary types of quantitative data—*numeric* and *categorical*—and each of these can be divided into a few subtypes. Numeric data is represented in numbers that indicate measurable quantities. It may be followed by some symbols to indicate units. Numeric data is further divided into *continuous data* and *discrete data*. With continuous data, the values can be any number. In other words, a value is chosen from an infinite set of numbers. With discrete data, the values follow a specific precision, which makes the set of possible values finite.

From the previous example, the class size 100, 150, etc. are numbers with the implied unit "students." Also, they indicate measurable quantities as they are head counts. Therefore, the class size is *numeric data*. It is also *continuous data* since the size numbers seem to be any natural numbers and these numbers are chosen from an infinite set of numbers, the set of natural numbers. Note that whether data is continuous (or discrete) also depends on the context. For example, the same class size data can be discrete if the campus enforces all classes to be 200 seats or less. Such restriction makes the class size values be chosen from a finite set of 200 numbers: 1, 2, 3, ..., 198, 199, 200.

Categorical data is represented in different forms such as words, symbols, and even numbers. A categorical value is chosen from a finite set of values, and the value does not necessarily indicate a measurable quantity. Categorical data can be divided into *nominal data* and *ordinal data*. For nominal data, the set of possible values does not include any ordering notion, whereas with ordinal data, the set of possible values includes an ordering notion.

The rest—semester, instructor, and ratings—are *categorical data*. They are represented in symbols (e.g., “Fall 2020,” “A”) or words (e.g., “Highly recommended”), and these values are chosen from the finite set of those symbols and words (e.g., A vs. B). The former two data are nominal since the semester and instructor do not have orders to follow, while the latter is ordinal since there is a notion of degree (Not recommended at all ~ Highly recommended). You may argue that the semester could have *chronological* ordering: Fall 2020 comes before Spring 2021, Fall 2021 follows Fall 2020. If you want to value that notion for your analysis, you could consider the semester data to be ordinal as well—the chronological ordering is indeed critical when you are looking at a time-series dataset. You will learn more about that in Time Series and Forecasting.

Example 1.2

Consider the jacket scenario in Example 1.1. In that example, we referred to two kinds of data:

1. The temperature during past three days—90°F, 85°F, ...
2. On each of those days, whether you needed a jacket—Yes, No, ...

What is the type of each data?

Answer

The temperatures are numbers followed by the unit degrees Fahrenheit (°F). Also, they indicate measurable quantities as they are specific readings from a thermometer. Therefore, the temperature is numeric data. They are also continuous data since they can be any real number, and the set of real numbers is infinite.

The other type of data—whether or not you needed a jacket—is categorical data. Categorical data are represented in symbols (Yes/No), and the values are chosen from the finite set of those symbols. They are also nominal since Yes/No does not have ordering to follow.

Datasets

A dataset is a collection of observations or data entities organized for analysis and interpretation, as shown in Table 1.1. Many datasets can be represented as a table where each row indicates a unique data entity and each column defines the structure of the entities.

Notice that the dataset we used in Table 1.1 has six *entities* (also referred to as items, *entries*, or *instances*), distinguished by semester. Each entity is defined by a combination of four attributes or characteristics (also known as *features* or *variables*)—Semester, Instructor, Class Size, and Rating. A combination of features characterizes an entry of a dataset.

Although the actual values of the attributes are different across entities, note that all entities have values for the same four attributes, which makes them a structured dataset. As a structured dataset, these items can be listed as a table where each item is listed along the rows of the table.

By contrast, an unstructured dataset is one that lacks a predefined or organized data model. While structured datasets are organized in a tabular format with clearly defined fields and relationships, unstructured data lacks a fixed schema. Unstructured data is often in the form of text, images, videos, audio recordings, or other content where the information doesn't fit neatly into rows and columns.

There are plenty of unstructured datasets. Indeed, some people argue there are more unstructured datasets than structured ones. A few examples include Amazon reviews on a set of products, Twitter posts last year, public images on Instagram, popular short videos on TikTok, etc. These unstructured datasets are often processed into a structured one so that data scientists can analyze the data. We'll discuss different data processing techniques in Collecting and Preparing Data.

Example 1.3

Let's revisit the jacket example: deciding whether to wear a jacket to class. Suppose the dataset looks as provided in Table 1.2:

Table 1.2 Jacket Dataset

Date	Temperature	Needed a Jacket?
Oct. 10	80°F	No
Oct. 11	60°F	Yes
Oct. 12	65°F	Yes
Oct. 13	75°F	No

Is this dataset structured or unstructured?

Answer

It is a structured dataset since 1) every individual item is in the same structure with the same three attributes—Date, Temperature, and Needed a Jacket—and 2) each value strictly fits into a cell of a table.

Example 1.4

How many entries and attributes does the dataset in the previous example have?

Answer

The dataset has four entries, each of which is identified with a specific date (Oct. 10, Oct. 11, Oct. 12, Oct. 13). The dataset has three attributes—Date, Temperature, Needed a Jacket.

Example 1.5

A dataset has a list of keywords that were searched on a web search engine in the past week. Is this dataset structured or unstructured?

Answer

The dataset is an unstructured dataset since each entry in the dataset can be a freeform text: a single word, multiple words, or even multiple sentences.

Example 1.6

The dataset from the previous example is processed so that now each search record is summarized as up to three words, along with the timestamp (i.e., when the search occurred). Is this dataset structured or unstructured?

Answer

It is a structured dataset since every entry of this dataset is in the same structure with two attributes: a short keyword along with the timestamp.

Dataset Formats and Structures (CSV, JSON, XML)

Datasets can be stored in different formats, and it's important to be able to recognize the most commonly used formats. This section covers three of the most often used formats for structured datasets—comma-separated values (CSV), JavaScript Object Notation (JSON), and Extensible Markup Language (XML). While CSV is the most intuitive way of encoding a tabular dataset, much of the data we would collect from the web (e.g., websites, mobile applications) is stored in the JSON or XML format. The reason is that JSON is the most suitable for exchanging data between a user and a server, and XML is the most suitable for complex dataset due to its hierarchy-friendly nature. Since they all store data as plain texts, you can open them using any typical text editors such as Notepad, Visual Studio Code, Sublime Text, or VI editor.

Table 1.3 summarizes the advantages and disadvantages of CSV, JSON, and XML dataset formats. Each of these is described in more detail below.

Table 1.3 Summary of the CSV, JSON, and XML formats

Dataset Format	Pros	Cons	Typical Use
CSV	Simple	- Difficult to add metadata - Difficult to parse if there are special characters - Flat structure	Tabular data
JSON	- Simple - Compatible with many languages - Easy to parse	- Difficult to add metadata - Cannot leave comments	Data that needs to be exchanged between a user and a server
XML	- Structured (so more readable) - Possible to add metadata	- Verbose - Complex structure with tags	Hierarchical data structures

Exploring Further: Popular and Reliable Databases to Search for Public Datasets

Multiple online databases offer public datasets for free. When you want to look for a dataset of interest, the following sources can be your initial go-to.

Government data sources include:

- [Data.gov](#)
- Bureau of Labor Statistics (BLS)
- National Oceanic and Atmospheric Administration (NOAA)
- [World Health Organization \(WHO\)](#)

Some reputable nongovernment data sources are:

- [Kaggle](#)
- [Statista](#)
- [Pew Research Center](#)

Comma-Separated Values (CSV)

The CSV stores each item in the dataset in a single line. Variable values for each item are listed all in one line, separated by commas (`,`). The previous example about signing up for a course can be stored as a CSV file. Figure 1.4 shows how the dataset looks when opened with a *text editor* (e.g., Notepad, TextEdit, MS Word, Google Doc) or programming software in the form of a *code editor* (e.g., Sublime Text, Visual Studio Code, XCode). Notice that commas are used to separate the attribute values within a single line (see Figure 1.4).

```
1 Semester,Instructor,Class Size,Rating
2 Fall 2020,A,100,Not recommended at all
3 Spring 2021,A,50,Highly recommended
4 Fall 2021,B,120,Not quite recommended
5 Spring 2022,B,40,Highly recommended
6 Fall 2022,A,110,Recommended
7 Spring 2023,B,50,Highly recommended
```

Figure 1.4 ch1-courseEvaluations.csv Opened with Visual Studio Code

Comma for New Line?

There is some flexibility on how to end a line with CSV files. It is acceptable to end with or without commas, as some software or programming languages automatically add a comma when generating a CSV dataset.

CSV files can be opened with spreadsheet software such as MS Excel and Google Sheets. The spreadsheet software visualizes CSV files more intuitively in the form of a table (see Figure 1.5). We will cover the basic use of Python for analyzing CSV files in Data Science with Python.

	A	B	C	D
1	Semester	Instructor	Class Size	Rating
2	Fall 2020	A	100	Not recommended at all
3	Spring 2021	A	50	Highly recommended
4	Fall 2021	B	120	Not quite recommended
5	Spring 2022	B	40	Highly recommended
6	Fall 2022	A	110	Recommended
7	Spring 2023	B	50	Highly recommended
8				

Figure 1.5 ch1-courseEvaluations.csv Opened with Microsoft Excel (Used with permission from Microsoft)

How to Download and Open a Dataset from the Ch1-Data Spreadsheet in This Text

A spreadsheet file accompanies each chapter of this textbook. The files include multiple tabs corresponding to a single dataset in the chapter. For example, the [spreadsheet file for this chapter](#) is shown in Figure 1.6. Notice that it includes multiple tabs with the names “ch1-courseEvaluations.csv,” “ch1-cancerdoc.csv,” and “ch1-iris.csv,” which are the names of each dataset.

	A	B	C	D	E	F	G	H
1	Semester	Instructor	Class Size	Rating				
2	Fall 2020	A	100	Not recommended at all				
3	Spring 2021	A	50	Highly recommended				
4	Fall 2021	B	120	Not quite recommended				
5	Spring 2022	B	40	Highly recom				
6	Fall 2022	A	110	Recommended				
7	Spring 2023	B	50	Highly recommended				
8								
9								

ch1-courseEvaluations.csv ch1-cancerdoc.csv ch1-iris.csv

Figure 1.6 The dataset spreadsheet file for Chapter 1 (Used with permission from Microsoft)

To save each dataset as a separate CSV file, choose the tab of your interest and select File > Save As ... > CSV File Format. This will only save the current tab as a CSV file. Make sure the file name is set correctly; it may have used the name of the spreadsheet file—“ch1-data.xlsx” in this case. You should name the generated CSV file as the name of the corresponding tab. For example, if you have generated a CSV file for the first tab of “ch1-data.xlsx,” make sure the generated file name is “ch1-courseEvaluations.csv.” This will prevent future confusion when following instructions in this textbook.

JavaScript Object Notation (JSON)

JSON uses the syntax of a programming language named JavaScript. Specifically, it follows JavaScript’s object syntax. Don’t worry, though! You do not need to know JavaScript to understand the JSON format.

Figure 1.7 provides an example of the JSON representation of the same dataset depicted in Figure 1.6.

```

1  {"Semester":"Fall 2020","Instructor":"A","Class Size":100,"Rating":"Not recommended at all"}
2  {"Semester":"Spring 2021","Instructor":"A","Class Size":50,"Rating":"Highly recommended"}
3  {"Semester":"Fall 2021","Instructor":"B","Class Size":120,"Rating":"Not quite recommended"}
4  {"Semester":"Spring 2022","Instructor":"B","Class Size":40,"Rating":"Highly recommended"}
5  {"Semester":"Fall 2022","Instructor":"A","Class Size":110,"Rating":"Recommended"}
6  {"Semester":"Spring 2023","Instructor":"B","Class Size":50,"Rating":"Highly recommended"}

```

Figure 1.7 CourseEvaluations.json Opened with Visual Studio Code

Notice that the JSON format starts and ends with a pair of curly braces (`{}`). Inside, there are multiple pairs of two fields that are separated by a colon (`:`). These two fields that are placed on the left and right of the colon are called a key and value, respectively, — key : value . For example, the dataset in Figure 1.7 has five pairs of key-values with the key "Semester": "Fall 2020" , "Semester": "Spring 2021" , "Semester": "Fall 2021" , "Semester": "Spring 2022" , "Semester": "Fall 2022" , and "Semester": "Spring 2023" .

[CourseEvaluations.json](#) has one key-value pair at the highest level: "Members": [...] . You can see that each item of the dataset is listed in the form of an *array* or list under the key "Members" . Inside the array, each item is also bound by curly braces and has a list of key-value pairs, separated by commas. Keys are used to describe attributes in the dataset, and values are used to define the corresponding values. For example, the first item in the JSON dataset above has four keys, each of which maps to each attribute—Semester, Instructor, Class Size, and Rating. Their values are "Fall 2020" , "A" , 100 , and "Not recommended at all" .

Extensible Markup Language (XML)

The XML format is like JSON, but it lists each item of the dataset using different symbols named *tags*. An XML tag is any block of text that consists of a pair of angle brackets (`<>`) with some text inside. Let's look at the example XML representation of the same dataset in Figure 1.8. Note that the screenshot of [CourseEvaluations.xml](#) below only includes the first three items in the original dataset.

```

<Members>
  <Evaluation>
    <Semester>"Fall 2020"</Semester>
    <Instructor>"A"</Instructor>
    <Classsize>100</Classsize>
    <Rating>"Not recommended at all"</Rating>
  </Evaluation>
  <Evaluation>
    <Semester>"Spring 2021"</Semester>
    <Instructor>"A"</Instructor>
    <Classsize>50</Classsize>
    <Rating>"Highly recommended"</Rating>
  </Evaluation>
  <Evaluation>
    <Semester>"Fall 2021"</Semester>
    <Instructor>"B"</Instructor>
    <Classsize>120</Classsize>
    <Rating>"Not quite recommended"</Rating>
  </Evaluation>
</Members>

```

Figure 1.8 ch1-courseEvaluations.xml with the First Three Entries Only, Opened with Visual Studio Code

[CourseEvaluations.xml](#) lists each item of the dataset between a pair of tags, `<members>` and `</members>` . Under `<members>` , each item is defined between `<evaluation>` and `</evaluation>` . Since the dataset in Figure 1.8 has three items, we can see three blocks of `<evaluation>` ... `</evaluation>` . Each item has four attributes, and they are defined as different XML tags as well— `<semester>` , `<instructor>` , `<classsize>` , and `<rating>` . They are also followed by closing tags such as `</semester>` , `</instructor>` , `</classsize>` , and `</rating>` .

[PubMed datasets](#) provides a list of articles that are published in the National Library of Medicine in XML format. Click Annual Baseline and download/open any .xml file. Note that all the .xml files are so big that they are compressed to .gz files. However, once you download one and attempt to open it by double-clicking, the file will automatically be decompressed and open. You will see a bunch of XML tags along with information about numerous publications, such as published venue, title, published date, etc.

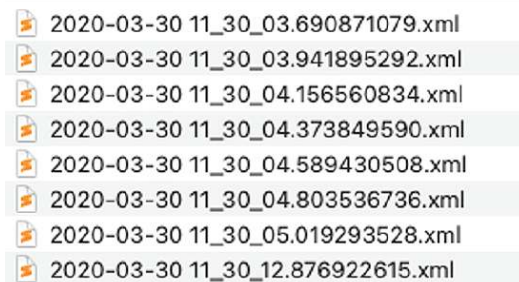
XML and Image Data

The XML format is also commonly used as an attachment to some image data. It is used to note supplementary information about the image. For example, the [Small Traffic Light Dataset](#) in Figure 1.9 comes with a set of traffic light images, placed in one of the three directories: `JPEGImages`, `train_images`, and `valid_images`. Each image directory is accompanied with another directory just for annotations such as `Annotations`, `train_annotations`, and `valid_annotations`.

```
├── Annotations
├── class.txt
├── JPEGImages
├── train_annotations
├── train_images
├── valid_annotations
└── valid_images
```

Figure 1.9 The Directory Structure of the Small Traffic Light Dataset

The annotation directories have a list of XML files, each of which corresponds to an image file with the same filename inside the corresponding image directory (Figure 1.10). Figure 1.11 shows that the first XML file in the Annotations directory includes information about the .jpg file with the same filename.



- 2020-03-30 11_30_03.690871079.xml
- 2020-03-30 11_30_03.941895292.xml
- 2020-03-30 11_30_04.156560834.xml
- 2020-03-30 11_30_04.373849590.xml
- 2020-03-30 11_30_04.589430508.xml
- 2020-03-30 11_30_04.803536736.xml
- 2020-03-30 11_30_05.019293528.xml
- 2020-03-30 11_30_12.876922615.xml

Figure 1.10 List of XML Files under the Annotations Directory in the Small Traffic Light Dataset
(source: “Small Traffic Light Dataset,” <https://www.kaggle.com/datasets/sovi...set-xml-format>)

```
<annotation>
  <folder/>
  <filename>2020-03-30_11_30_03.690871079.jpg</filename>
  <database/>
  <annotation/>
  <image/>
  <size>
    <height>1080</height>
    <width>1920</width>
    <depth>3</depth>
  </size>
  <segmented/>
  <object>
    <name>green</name>
    <pose/>
    <truncated/>
    <difficult/>
    <bndbox>
      <xmin>616</xmin>
      <ymin>477</ymin>
      <xmax>633</xmax>
      <ymax>521</ymax>
    </bndbox>
  </object>
</annotation>
```

Figure 1.11 2020-03-30_11_30_03.690871079.xml, an Example XML file within the Small Traffic Light Dataset (Source: “Small Traffic Light Dataset,” <https://www.kaggle.com/datasets/sovi...set-xml-format>)

JSON and XML Dataset Descriptions

Both JSON and XML files often include some description(s) of the dataset itself as well (known as *metadata*), and they are included as a separate entry in the file ({ } or <>). In Figure 1.12 and Figure 1.13, the actual data entries are listed inside “itemData” and <data> , respectively. The rest are used to provide background information on the dataset. For example:

- “creationDateTime”: describes when the dataset was created.
- <name> is used to write the name of this dataset.
- <metadata> is used to describe each column name of the dataset along with its data type.

```
{
  "creationDateTime": "2023-03-22T11:53:27",
  "datasetJSONVersion": "1.0.0",
  "fileOID": "www.sponsor.org.project123.final",
  "sourceSystemVersion": "1.2.3",
  "clinicalData": {
    "items": [
      {"OID": "ITEMGROUPDATASEQ", "name": "ITEMGROUPDATASEQ", "label": "Record identifier", "type": "integer"},
      {"OID": "IT.STUDYID", "name": "STUDYID", "label": "Study identifier", "type": "string"},
      {"OID": "IT.DOMAIN", "name": "DOMAIN", "label": "Domain identifier", "type": "string", "length": 2},
    ],
    "itemData": [
      [1, "MyStudy1", "D1"],
      [2, "MyStudy2", "D2"],
      ...
    ]
  }
}
```

Figure 1.12 An Example JSON File with Metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <name>CEREALS</name>
  <metadata>
    <version>1.2</version>
    <date>04/02/2024</date>
    <description>a list of some popular cereals</description>
    <col name="NAME" type="string"/>
    <col name="MANUFACTURER" type="string"/>
    <col name="CALORIES_PER_SERVING" type="integer"/>
  </metadata>
  <data>
    <row>
      <value>Cheerios</value>
      <value>General Mills</value>
      <value>140</value>
    </row>
    <row>
      <value>Corn Flakes</value>
      <value>Kellogg's</value>
      <value>150</value>
    </row>
  </data>
</dataset>
```

Figure 1.13 An Example XML Dataset with Metadata

The [Face Mask Detection](#) dataset has a set of images of human faces with masks on. It follows a similar structure as well. The dataset consists of two directories—annotations and images. The former is in the XML format. The name of each XML file includes any text description about the image with the same filename. For example, “maksssksksss0.xml” includes information on “maksssksksss0.png.”

Example 1.7

The Iris Flower dataset ([ch1-iris.csv](#)) is a classic dataset in the field of data analysis.¹ Download this dataset and open it with a code editor (e.g., Sublime Text, XCode, Visual Studio Code). (We recommend that if you do not have any code editor installed, you install one. All three of these editors are quick and easy to install.) Now, answer these questions:

- How many items are there in the dataset?
- How many attributes are there in the dataset?
- What is the second attribute in the dataset?

Answer

There are 151 rows in the dataset with the header row at the top, totaling 150 items. There are five attributes listed across columns: `sepal_length`, `sepal_width`, `petal_length`, `petal_width`, `species`. The second attribute is `sepal_width`.

Example 1.8

The Jeopardy dataset ([ch1-jeopardy.json](#)) is formatted in JSON. Download and open it with a code editor (e.g., Notepad, Sublime Text, Xcode, Visual Studio Code).

- How many items are there in the dataset?
- How many attributes are there in the dataset?
- What is the third item in the dataset?

Answer

There are 409 items in the dataset, and each item has seven attributes: “`category`”, “`air_date`”, “`question`”, “`value`”, “`answer`”, “`round`”, and “`show_number`”. The third item is located at index 2 of the first list as shown in Figure 1.14.

```
"category" : string "EVERYBODY TALKS ABOUT IT..."
"air_date" : string "2004-12-31"
"question" :
string "'The city of Yuma in this state has a record average of 4,055 hours of sunshine each year'"
"value" : string "$200"
"answer" : string "Arizona"
"round" : string "Jeopardy!"
"show_number" : string "4680"
```

Figure 1.14 The Third Item in the Jeopardy Dataset

Footnotes

- 1The Iris Flower dataset was introduced by the British statistician and biologist Ronald Fisher in his 1936 paper “The Use of Multiple Measurements in Taxonomic Problems.” This work became a landmark study in the use of multivariate data in classification problems and frequently makes an appearance in data science as a convenient test case for machine learning and neural network algorithms. The Iris Flower dataset is often used as a beginner’s dataset to demonstrate various techniques, such as classification of algorithms, formatted in CSV.

This page titled [1.4: Data and Datasets](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.5: Using Technology for Data Science

Learning Objectives

By the end of this section, you should be able to:

- 1.4.1 Explain how statistical software can help with data analysis.
- 1.4.2 Explain the uses of different programs and programming languages for data manipulation, analysis, and visualizations.
- 1.4.3 Explain the uses of various data analysis tools used in data science applications.

Technology empowers data analysts, researchers, and organizations to leverage data, extract actionable insights, and make decisions that optimize processes and improve outcomes in many areas of our lives. Specifically, technology provides the tools, platforms, and algorithms that enable users to efficiently process and analyze data—especially complex datasets. The choice of technology used for a data science project will vary depending on the goals of the project, the size of the datasets, and the kind of analysis required.

Spreadsheet Programs

Spreadsheet programs such as Excel and Google Sheets are software applications consisting of electronic worksheets with rows and columns where data can be entered, manipulated, and calculated. Spreadsheet programs offer a variety of functions for data manipulation and can be used to easily create charts and tables. Excel is one of the most widely used spreadsheet programs, and as part of Microsoft Office, it integrates well with other Office products. Excel was first released by Microsoft in 1987, and it has become one of the most popular choices for loading and analyzing tabular data in a spreadsheet format. You are likely to have used Excel in some form or other—perhaps to organize the possible roommate combinations in your dorm room or to plan a party, or in some instructional context. We refer to the use of Excel to manipulate data in some of the examples of this text because sometimes a spreadsheet is simply the easiest way to work with certain datasets. (See Appendix A: Review of Excel for Data Science for a review of Excel functionality.)

Google Sheets is a cloud-based spreadsheet program provided by Google as part of the Google Workspace. Because it is cloud-based, it is possible to access spreadsheets from any device with an internet connection. This accessibility allows for collaboration and real-time updates among multiple users, enhancing communication within a team and making it ideal for team projects or data sharing among colleagues. Users can leave comments, track changes, and communicate within the spreadsheet itself.

The user interfaces for both Excel and Google Sheets make these programs very user-friendly for many applications. But these programs have some limitations when it comes to large databases or complex analyses. In these instances, data scientists will often turn to a programming language such as Python, R, or SPSS.

Programming Languages

A programming language is a formal language that consists of a set of instructions or commands used to communicate with a computer and to instruct it to perform specific tasks that may include data manipulation, computation, and input/output operations. Programming languages allow developers to write algorithms, create software applications, and automate tasks and are better suited than spreadsheet programs to handle complex analyses.

Python and R are two of the most commonly used programming languages today. Both are open-source programming languages. While Python started as a general-purpose language that covers various types of tasks (e.g., numerical computation, data analysis, image processing, and web development), R is more specifically designed for statistical computing and graphics. Both use simple syntax compared to conventional programming languages such as Java or C/C++. They offer a broad collection of packages for data manipulation, statistical analysis, visualization, machine learning, and complex data modeling tasks.

We have chosen to focus on Python in this text because of its straightforward and intuitive syntax, which makes it especially easy for beginners to learn and apply. Also, Python skills can apply to a wide range of computing work. Python also contains a vast network of libraries and frameworks specifically designed for data analysis, machine learning, and scientific computing. As we'll see, Python libraries such as [NumPy](#), [Pandas](#), [Matplotlib](#), and [Seaborn](#) provide powerful tools for data manipulation, visualization,

and machine learning tasks, making Python a versatile choice for handling datasets. You'll find the basics of R and various R code examples in Appendix B: Review of R Studio for Data Science. If you want to learn how to use R, you may want to practice with [RStudio](#), a commonly used software application to edit/run R programs.

Python in Excel

Microsoft recently launched a new feature for Excel named “Python in Excel.” This feature allows a user to run Python code to analyze data directly in Excel. This textbook does not cover this feature, but instead presents Python separately, as it is also crucial for you to know how to use each tool in its more commonly used environment. If interested, refer to [Microsoft's announcement](#).

Exploring Further: Specialized Programming Languages

Other programming languages are more specialized for a particular task with data. These include SQL, Scala, and Julia, among others, as briefly described in this article on “[The Nine Top Programming Languages for Data Science](#).”

Other Data Analysis/Visualization Tools

There are a few other data analysis tools that are strong in data visualization. [Tableau](#) and [PowerBI](#) are user-friendly applications for data visualization. They are known for offering more sophisticated, interactive visualizations for high-dimensional data. They also offer a relatively easy user interface for compiling an analysis dashboard. Both allow users to run a simple data analysis as well before visualizing the results, similar to Excel.

In general, data visualization aims to make complex data more understandable and usable. The tools and technology described in this section offer a variety of ways to go about creating visualizations that are most accessible. Refer to Visualizing Data for a deeper discussion of the types of visualizations—charts, graphs, boxplots, histograms, etc.—that can be generated to help find the meaning in data.

Exploring Further: Evolving Professional Standards

Data science is a field that is changing daily; the introduction of artificial intelligence (AI) has increased this pace. Technological, social, and ethical challenges with AI are discussed in Natural Language Processing, and ethical issues associated with the whole data science process, including the use of machine learning and artificial intelligence, are covered in Ethics Throughout the Data Science Cycle. A variety of data science professional organizations are working to define and update process and ethical standards on an ongoing basis. Useful references may include the following:

- [Initiative for Analytics and Data Science Standards \(IADSS\)](#)
- Data Science Association (DSA)
- Association of Data Scientists (ADaSci)

This page titled [1.5: Using Technology for Data Science](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.6: Correlation and Linear Regression Analysis

Learning Objectives

By the end of this section, you should be able to:

- 4.3.1 Create scatterplots and calculate and interpret correlation coefficients.
- 4.3.2 Perform linear regression and determine the best-fit linear equation.
- 4.3.3 Use Python to calculate correlation coefficients and determine equations of linear regression models.

We briefly introduced correlation analysis at the beginning of this chapter, but now we want to dig in a little deeper. Data scientists are often interested in knowing if there are relationships, or a *correlation*, between two numeric quantities. For example, a medical researcher might be interested in investigating if there is a relationship between age and blood pressure. A college professor might be interested to know if there is a relationship between time spent on social media by students and corresponding academic grades.

Correlation analysis allows for the determination of a statistical relationship between two numeric quantities, or variables—an independent variable and a dependent variable. The independent variable is the variable that you can change or control, while the dependent variable is what you measure or observe to see how it responds to those changes. Regression analysis takes this a step further by quantifying the relationship between the two variables, and it can be used to predict one quantity based on a second quantity, assuming there is a significant correlation between the two quantities. For example, the value of a car can be predicted based on the age of the car.

Earlier we noted many business-related applications where correlation and regression analysis are used. For instance, regression analysis can be used to establish a mathematical equation that relates a dependent variable (such as sales) to an independent variable (such as advertising expenditure). But it has many applications apart from business as well. An automotive engineer is interested in the correlation between outside temperature and battery life for an electric vehicle, for instance.

Our discussion here will focus on linear regression—analyzing the relationship between one dependent variable and one independent variable, where the relationship can be modeled using a linear equation.

Scatterplots and Correlation

In correlation analysis, we study the relationship between bivariate data, which is data collected on two variables where the data values are paired with one another. Correlation measures the association between two numeric variables. We may be interested in knowing if there is a correlation between bond prices and interest rates or between the age of a car and the value of the car. To investigate the correlation between two numeric quantities, the first step is to collect (x, y) data for the two numeric quantities of interest and then create a scatterplot that will graph the (x, y) ordered pairs. The independent, or explanatory, quantity is labeled the x -variable, and the dependent, or response, quantity is labeled the y -variable.

For example, let's say that a financial analyst wants to know if the price of Nike stock is correlated with the value of the S&P 500 (Standard & Poor's 500 stock market index). To investigate this, monthly data can be collected for Nike stock prices and value of the S&P 500 for a period of time, and a scatterplot can be created and examined. A scatterplot, or scatter diagram, is a graphical display intended to show the relationship between two variables. The setup of the scatterplot is that one variable is plotted on the horizontal axis and the other variable is plotted on the vertical axis. Each pair of data values is considered as an Math input error

Date	S&P 500 (x)	Nike Stock Price (y)
Month 1	2912.43	87.18
Month 2	3044.31	98.58
Month 3	3100.29	98.05
Month 4	3271.12	97.61
Month 5	3500.31	111.89
Month 6	3363.00	125.54
Month 7	3269.96	120.08
Month 8	3621.63	134.70
Month 9	3756.07	141.47
Month 10	3714.24	133.59
Month 11	3811.15	134.78
Month 12	3943.34	140.45

Table 4.14 Nike Stock Price (\$) and Value of S&P 500 over a One-Year Time Period
(source: Yahoo! Finance)

To assess linear correlation, examine the graphical trend of the data points on the scatterplot to determine if a straight-line pattern exists (see Figure 4.5). If a linear pattern exists, the correlation may indicate either a positive or a negative correlation. A positive correlation indicates that as the independent variable increases, the dependent variable tends to increase as well, or as the independent variable decreases, the dependent variable tends to decrease (the two quantities move in the same direction). A negative correlation indicates that as the independent variable increases, the dependent variable decreases, or as the independent variable decreases, the dependent variable increases (the two quantities move in opposite directions). If there is no relationship or association between the two quantities, where one quantity changing does not affect the other quantity, we conclude that there is no correlation between the two variables.

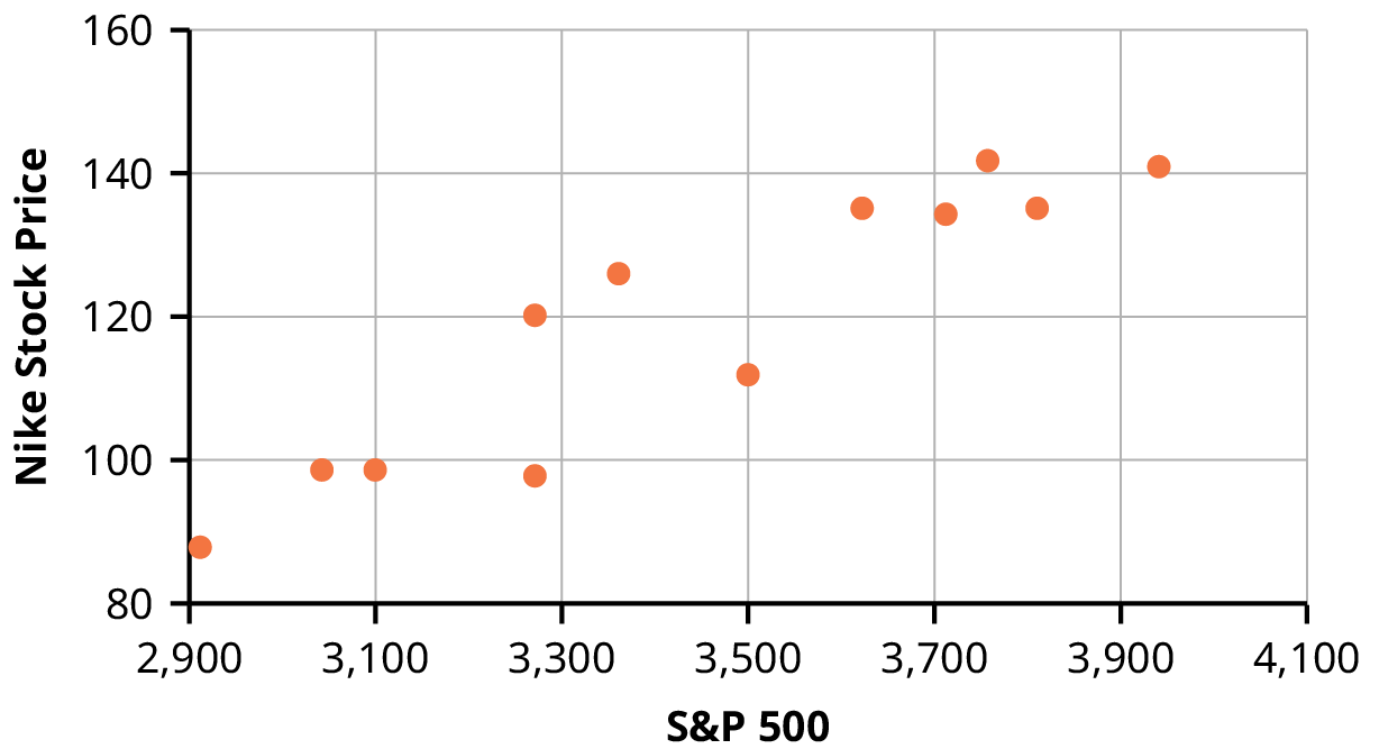


Figure 4.5 Scatterplot of Nike Stock Price (\$) and Value of S&P 500
(source: Yahoo! Finance)

From the scatterplot in the Nike stock versus S&P 500 example, we note that the trend reflects a positive correlation in that as the value of the S&P 500 increases, the price of Nike stock tends to increase as well.

When inspecting a scatterplot, it may be difficult to assess a correlation based on a visual inspection of the graph alone. A more precise assessment of the correlation between the two quantities can be obtained by calculating the numeric correlation coefficient (referred to using the symbol r).

The correlation coefficient is a measure of the strength and direction of the correlation between the independent variable x and the dependent variable y .

The formula for r is shown; however, software is typically used to calculate the correlation coefficient.

$$r = \frac{n\sum xy - (\sum x)(\sum y)}{\sqrt{n\sum x^2 - (\sum x)^2} \sqrt{n\sum y^2 - (\sum y)^2}}$$

where n refers to the number of data pairs and the symbol $\sum x$ indicates to sum the x -values.

Note that this value of “ r ” is sometimes referred to as the “Pearson correlation coefficient.”

Table 4.15 provides a step-by-step procedure on how to calculate the correlation coefficient r .

Table 4.15 Steps for Calculating the Correlation Coefficient

Step	Representation in Symbols
1. Calculate the sum of the x -values.	Σx
2. Calculate the sum of the y -values.	Σy
3. Multiply each x -value by the corresponding y -value and calculate the sum of these xy products.	Σxy
4. Square each x -value and then calculate the sum of these squared values.	Σx^2
5. Square each y -value and then calculate the sum of these squared values.	Σy^2
6. Determine the value of n , which is the number of data pairs.	n
7. Use these results to then substitute into the formula for the correlation coefficient.	$r = \frac{n\Sigma xy - (\Sigma x)(\Sigma y)}{\sqrt{n\Sigma x^2 - (\Sigma x)^2} \sqrt{n\Sigma y^2 - (\Sigma y)^2}}$

Note that since r is calculated using sample data, r is considered a *sample statistic* and is used to measure the strength of the correlation for the two population variables. Recall that sample data indicates data based on a subset of the entire population.

As mentioned, given the complexity of this calculation, software is typically used to calculate the correlation coefficient. There are several options for calculating the correlation coefficient in Python. The example shown uses the `scipy.stats` library in Python which includes the function `pearsonr()` for calculating the Pearson correlation coefficient (r).

We will create two numerical arrays in Python using the `np.array()` function and then use the `pearsonr()` function to calculate the Pearson correlation coefficient for the `Math input error`

The following Python code provides the correlation coefficient for the Nike Stock Price dataset as 0.923.

Python Code

```
# import libraries
import numpy as np
from scipy.stats import pearsonr

# establish x and y data, x-data is S&P500 value, y-data is Nike Stock Price
SP500 = np.array([2912.43, 3044.31, 3100.29, 3271.12, 3500.31, 3363.00, 3260.00])
Nike = np.array([87.18, 98.58, 98.05, 97.61, 111.89, 125.54, 120.08, 134.76])

#pearson r returns both value of r and corresponding p-value
r, p = pearsonr(SP500, Nike)

#print value of r , rounded to 3 decimal places

print("Correlation coefficient: ", round(r, 3))
```

The resulting output will look like this:

Correlation coefficient: 0.923

Interpret a Correlation Coefficient

Once the value of r is calculated, this measurement provides two indicators for the correlation:

- the strength of the correlation based on the *value* of r
- the direction of the correlation based on the *sign* of r

The *value* of r gives us this information:

- The value of r is always between -1 and $+1$: $-1 \leq r \leq 1$.
- The size of the correlation r indicates the strength of the linear relationship between the two variables. Values of r close to -1 or to $+1$ indicate a stronger linear relationship.
- If $r = 0$, there is no linear relationship between the two variables (*no linear correlation*).
- If $r = 1$, there is perfect positive correlation. If $r = -1$, there is perfect negative correlation. In both of these cases, all the original data points lie on a straight line.

The sign of r gives us this information:

- A positive value of r means that when x increases, y tends to increase, and when x decreases, y tends to decrease (*positive correlation*).
- A negative value of r means that when x increases, y tends to decrease, and when x decreases, y tends to increase (*negative correlation*).

From the statistical results shown in the Python output, the correlation coefficient r is 0.923, which indicates that the relationship between Nike stock and the value of the S&P 500 over this time period represents a strong, positive correlation.

Test a Correlation Coefficient for Significance

The correlation coefficient, r , tells us about the strength and direction of the linear relationship between x and y . The sample data are used to compute r , the correlation coefficient for the sample. If we had data for the entire population (that is, all measurements of interest), we could find the population correlation coefficient, which is labeled as the Greek letter ρ (pronounced “rho”). But because we have only sample data, we cannot calculate the population correlation coefficient. The sample correlation coefficient, r , is our estimate of the unknown population correlation coefficient.

- ρ = population correlation coefficient (unknown)
- r = sample correlation coefficient (known; calculated from sample data)

An important step in the correlation analysis is to determine if the correlation is significant. By this, we are asking if the correlation is strong enough to allow meaningful predictions for y based on values of x . One method to test the significance of the correlation is to employ a hypothesis test. The hypothesis test lets us decide whether the value of the population correlation coefficient ρ is close to zero or significantly different from zero. We decide this based on the sample correlation coefficient r and the sample size n .

If the test concludes that the correlation coefficient is significantly different from zero, we say that the correlation coefficient is significant.

- Conclusion: There is sufficient evidence to conclude that there is a significant linear relationship between the two variables because the correlation coefficient is significantly different from zero.
- What the conclusion means: There is a significant linear relationship between the two variables. If the test concludes that the correlation coefficient is not significantly different from zero (it is close to zero), we say that correlation coefficient is not significant.

A hypothesis test can be performed to test if the correlation is significant. A hypothesis test is a statistical method that uses sample data to test a claim regarding the value of a population parameter. In this case, the hypothesis test will be used to test the claim that the population correlation coefficient ρ is equal to zero.

Use these hypotheses when performing the hypothesis test:

- Null hypothesis: $H_0 : \rho = 0$
- Alternate hypothesis: $H_a : \rho \neq 0$

The hypotheses can be stated in words as follows:

- Null hypothesis H_0 : The population correlation coefficient is *not* significantly different from zero. There is *not* a significant linear relationship (correlation) between x and y in the population.
- Alternate hypothesis H_a : The population correlation coefficient is significantly different from zero. There is a significant linear relationship (correlation) between x and y in the population.

A quick shorthand way to test correlations is the relationship between the sample size and the correlation. If $|r| \geq \frac{2}{\sqrt{n}}$, then this implies that the correlation between the two variables demonstrates that a linear relationship exists and is statistically significant at approximately the 0.05 level of significance. As the formula indicates, there is an inverse relationship between the sample size and the required correlation for significance of a linear relationship. With only 10 observations, the required correlation for significance is 0.6325; for 30 observations, the required correlation for significance decreases to 0.3651; and at 100 observations, the required level is only 0.2000.

Interpreting the Significance of the Sample Correlation Coefficient

- If r is significant and the scatterplot shows a linear trend, the line can be used to predict the value of y for values of x that are within the domain of observed x -values.
- If r is not significant OR if the scatterplot does not show a linear trend, the line should *not* be used for prediction.
- If r is significant and the scatterplot shows a linear trend, the line may *not* be appropriate or reliable for prediction *outside* the domain of observed x -values in the data.

Example 4.22

Suppose that the chief financial officer (CFO) of a corporation is investigating the correlation between stock prices and unemployment rate over a period of 10 years and finds the correlation coefficient to be -0.68 . There are 10 (x, y) data points in the dataset. Should the CFO conclude that the correlation is significant for the relationship between stock prices and unemployment rate based on a level of significance of 0.05?

Answer

When using a level of significance of 0.05, if $|r| \geq \frac{2}{\sqrt{n}}$, then this implies that the correlation between the two variables demonstrates a linear relationship that is statistically significant at approximately the 0.05 level of significance. In this example, we check if $|-0.68|$ is greater than or equal to $\frac{2}{\sqrt{n}}$ where $n = 10$. Since $\frac{2}{\sqrt{10}}$ is approximately 0.632, this indicates that the r -value of -0.68 is greater than $\frac{2}{\sqrt{n}}$, and thus the correlation is deemed significant.

Linear Regression

A regression model is typically developed once the correlation coefficient has been calculated and a determination has been made that the correlation is significant. The regression model can then be used to predict one quantity (the dependent variable) based on the other (the independent variable). For example, a business may want to establish a correlation between the amount the company spent on advertising (the independent variable) versus its recorded sales (the dependent variable). If a strong enough correlation is established, then the business manager can predict sales based on the amount spent on advertising for a given period. In this discussion we will focus on linear regression, where a straight line is used to model the relationship between the two variables. Once a straight-line model is developed, this model can then be used to predict the value of the dependent variable for a specific value of the independent variable.

Method of Least Squares and Residuals

To create a linear model that best fits the (x, y) points on a scatterplot, researchers generally use the least squares method based on a mathematical algorithm designed to minimize the squared distances from the (x, y) data points to the fitted line.

Recall from algebra that the equation of a straight line is given by:

$$y = mx + b$$

where m is the slope of the line and b is the y -intercept of the line.

The slope measures the steepness of the line, and the y -intercept is that point on the y -axis where the graph crosses, or intercepts, the y -axis.

In linear regression analysis, the equation of the straight line is written in a slightly different way using the model

$$\hat{y} = a + bx$$

In this format, b is the slope of the line, and a is the y -intercept of the line. The notation $\hat{y}$ is called y -hat and is used to indicate a predicted value of the dependent variable y for a certain value of the independent variable x .

If a line extends uphill from left to right, the slope is a positive value (see Figure 4.6; if the line extends downhill from left to right, the slope is a negative value).

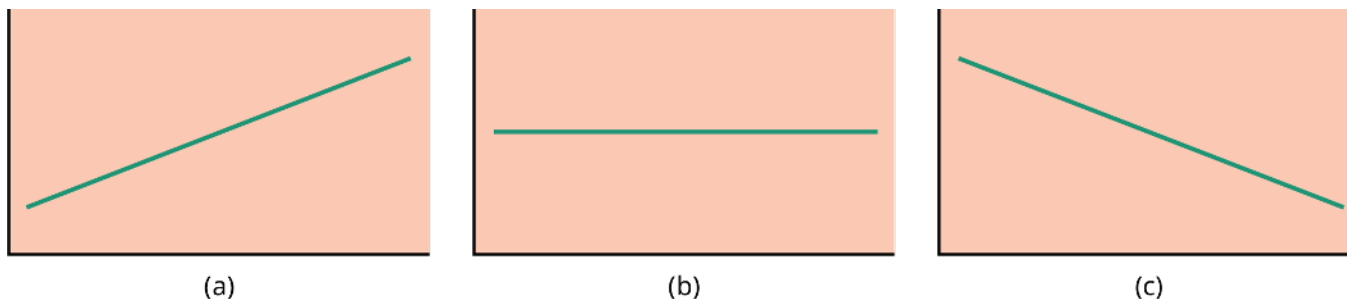


Figure 4.6 Three examples of graphs of $\hat{y} = a + bx$. (a) If $b > 0$, the line slopes upward to the right. (b) If $b = 0$, the line is horizontal. (c) If $b < 0$, the line slopes downward to the right.

When generating the equation of a line in algebra using $y = mx + b$, two (x, y) points were required to generate the equation. However, in regression analysis, all the (x, y) points in the dataset will be utilized to develop the linear regression model.

The first step in any regression analysis is to create the scatterplot. Then proceed to calculate the correlation coefficient r and check this value for significance. If we think that the points show a linear relationship, we would like to draw a line on the scatterplot. This line can be calculated through a process called linear regression. However, we only calculate a regression line if one of the variables helps to explain or predict the other variable. If x is the independent variable and y the dependent variable, then we can use a regression line to predict y for a given value of x .

As an example of a regression equation, assume that a correlation exists between the monthly amount spent on advertising and the monthly revenue for a Fortune 500 company. After collecting (x, y) data for a certain time period, the company determines the regression equation is of the form

$$\hat{y} = 9376.7 + 61.8x$$

where x represents the monthly amount spent on advertising (in thousands of dollars), $\hat{y}$ represents the monthly revenues for the company (in thousands of dollars), the slope is 61.8, and the y -intercept is 9376.7.

A scatterplot of the Math input error

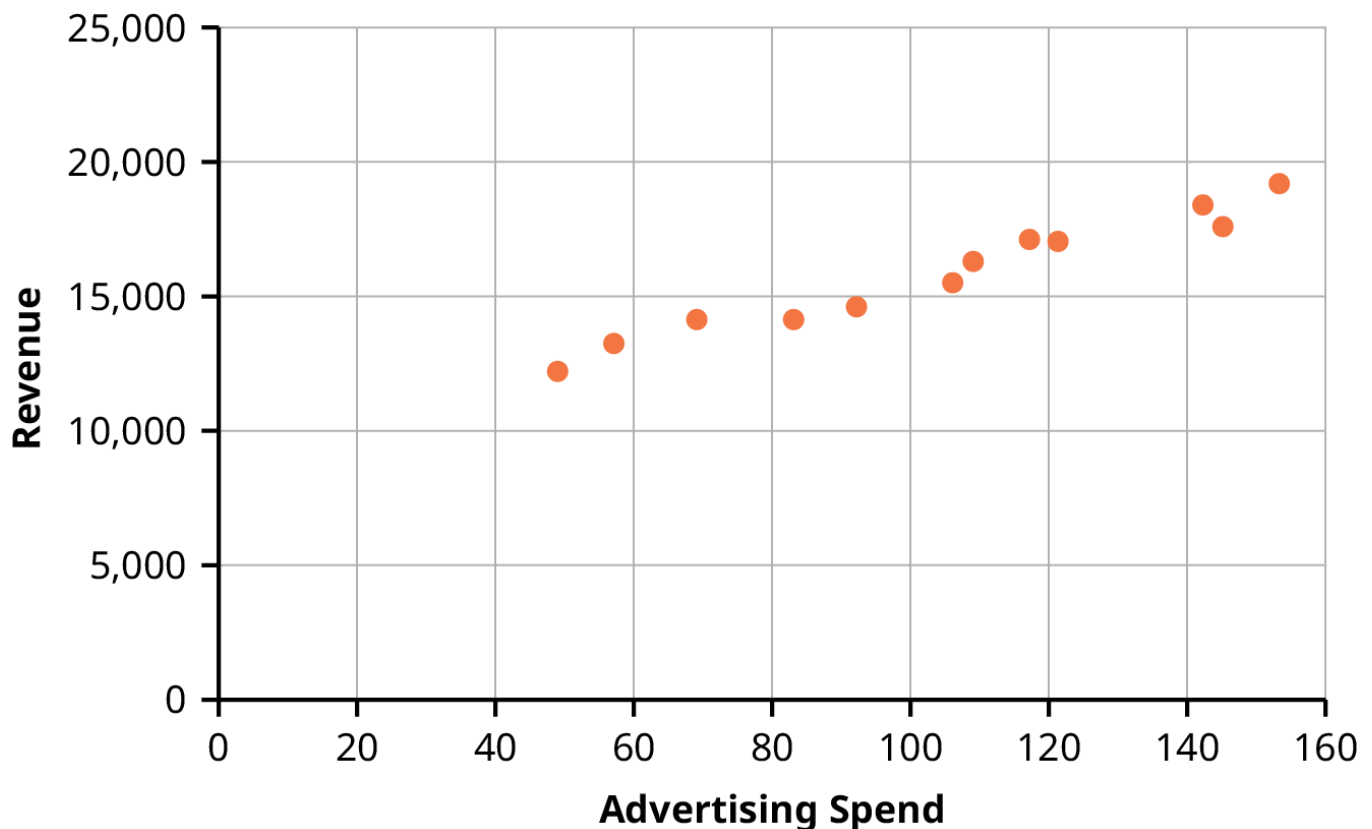


Figure 4.7 Scatterplot of Revenue versus Advertising for a Fortune 500 Company (\$000s)

The company would like to predict the monthly revenue if its executives decide to spend \$150,000 on advertising next month. To determine the estimate of monthly revenue, let $x = 150$ in the regression equation and calculate a corresponding value for $\hat{y}$:

$$\begin{aligned}\hat{y} &= 9376.7 + 61.8x \\ \hat{y} &= 9376.7 + 61.8(150) \\ \hat{y} &= 18646.7\end{aligned}$$

This predicted value of y indicates that the anticipated revenue would be \$18,646,700, given the advertising spend of \$150,000.

Notice that from past data, there may have been a month where the company actually did spend \$150,000 on advertising, and thus the company may have an actual result for the monthly revenue. This actual, or observed, amount can be compared to the prediction from the linear regression model to calculate what is called a residual.

A residual is the difference between an observed y -value and the predicted y -value obtained from the linear regression equation. As an example, assume that in a previous month, the actual monthly revenue for an advertising spend of \$150,000 was \$19,200,000, and thus $y = 19,200$. The residual for this data point can be calculated as follows:

$$\begin{aligned}\text{Residual} &= (\text{observed } y\text{-value}) - (\text{predicted } y\text{-value}) \\ \text{Residual} &= y - \hat{y} \\ \text{Residual} &= 19200 - 18646.7 = 553.3\end{aligned}$$

Notice that residuals can be positive, negative, or zero. If the observed y -value exactly matches the predicted y -value, then the residual will be zero. If the observed y -value is greater than the predicted y -value, then the residual will be a positive value. If the observed y -value is less than the predicted y -value, then the residual will be a negative value.

When formulating the best-fit linear regression line to the points on the scatterplot, the mathematical analysis generates a linear equation where the sum of the squared residuals is minimized. This analysis is referred to as the method of least squares. The result is that the analysis generates a linear equation that is the “best fit” to the points on the scatterplot, in the sense that the line minimizes the differences between the predicted values and observed values for y ; this is generally referred to as the best-fit linear equation.

Example 4.23

Suppose that the chief financial officer of a corporation has created a linear model for the relationship between the company stock and interest rates. When interest rates are at 5%, the company stock has a value of \$94. Using the linear model, when interest rates are at 5%, the model predicts the value of the company stock to be \$99. Calculate the residual for this data point.

Answer

A residual is the difference between an observed y -value and the predicted y -value obtained from the linear regression equation:

$$\begin{aligned}\text{Residual} &= (\text{observed } y\text{-value}) - (\text{predicted } y\text{-value}) \\ \text{Residual} &= y - \hat{y} \\ \text{Residual} &= 94 - 99 = -5\end{aligned}$$

The goal in regression analysis is to determine the coefficients a and b in the following regression equation:

$$\hat{y} = a + bx$$

Once the (x, y) has been collected, the slope (b) and y -intercept (a) can be calculated using the following formulas:

$$\begin{aligned}b &= \frac{n\sum xy - (\sum x)(\sum y)}{n\sum x^2 - (\sum x)^2} \\ a &= \frac{\sum y}{n} - b\frac{\sum x}{n}\end{aligned}$$

where n refers to the number of data pairs and $\sum x$ indicates sum of the x -values.

Notice that the formula for the y -intercept requires the use of the slope result (b), and thus the slope should be calculated first, and the y -intercept should be calculated second.

When making predictions for y , it is always important to plot a scatter diagram first. If the scatterplot indicates that there is a linear relationship between the variables, then it is reasonable to use a best-fit line to make predictions for y , given x is within the domain of x -values in the sample data, *but not necessarily for x -values outside that domain*.

Using Technology for Linear Regression

Typically, technology is used to calculate the best-fit linear model as well as calculate correlation coefficients and scatterplot. Details of using Python for these calculations are provided in Using Python for Correlation and Linear Regression.

Assumptions for Linear Regression

Testing the significance of the correlation coefficient requires that certain assumptions about the data are satisfied. The premise of this test is that the data are a sample of observed points taken from a larger population. We have not examined the entire population because it is not possible or feasible to do so. We are examining the sample to draw a conclusion about whether the linear relationship that we see between x and y data in the sample data provides strong enough evidence that we can conclude that there actually is a linear relationship between x and y data in the population.

The regression line equation that we calculate from the sample data gives the best-fit line for our particular sample. We want to use this best-fit line for the sample as an estimate of the best-fit line for the population (Figure 4.8). Examining the scatterplot and testing the significance of the correlation coefficient helps us determine if it is appropriate to do this.

These are the assumptions underlying the test of significance:

1. There is a linear relationship in the population that models the average value of y for varying values of x . In other words, the expected value of y for each particular value lies on a straight line in the population. (We do not know the equation for the line for the population. Our regression line from the sample is our best estimate of this line in the population.)
2. The y -values for any particular x -value are normally distributed about the line. This implies that there are more y -values scattered closer to the line than are scattered farther away. Assumption (1) implies that these normal distributions are centered on the line: the means of these normal distributions of y -values lie on the line.

3. The standard deviations of the population y -values about the line are equal for each value of x . In other words, each of these normal distributions of y -values has the same shape and spread about the line.
4. The residual errors are mutually independent (no pattern).
5. The data are produced from a well-designed random sample or randomized experiment.

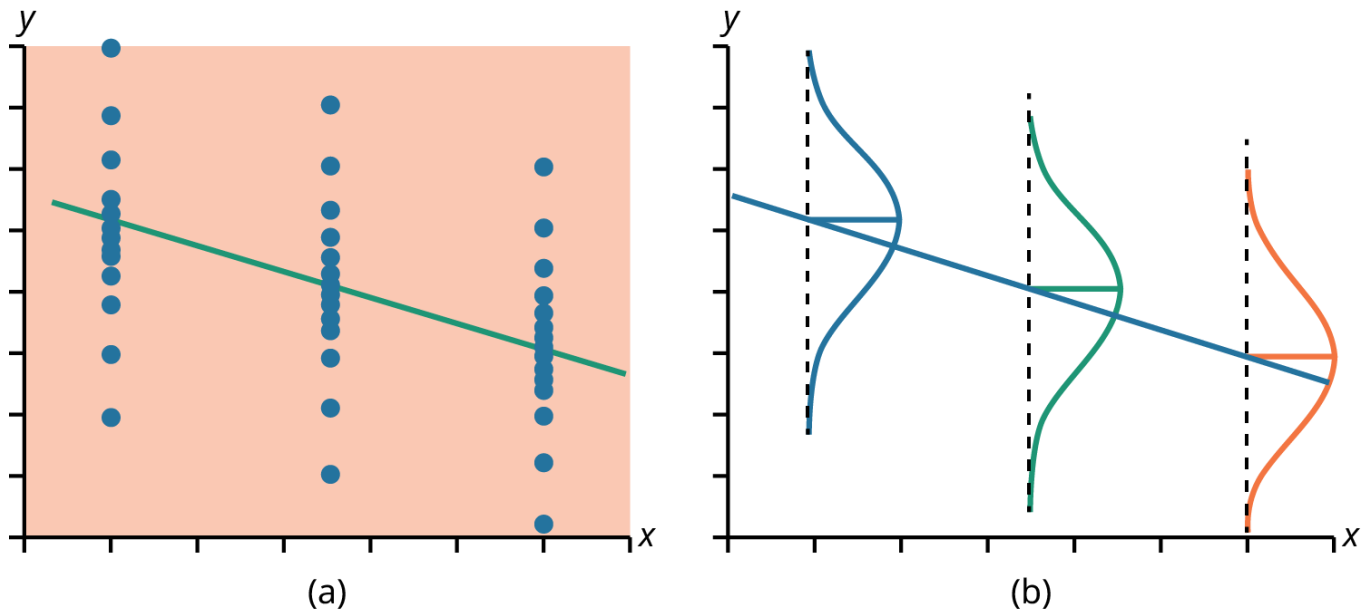


Figure 4.8 Best-Fit Line. Note that the y -values for each x -value are normally distributed about the line with the same standard deviation. For each x -value, the mean of the y -values lies on the regression line. More y -values lie near the line than are scattered farther away from the line

Using Python for Correlation and Linear Regression

Once a correlation has been deemed significant, a linear regression model is developed. The goal in the regression analysis is to determine the coefficients a and b in the following regression equation:

$$\hat{y} = a + bx$$

These formulas can be quite cumbersome, especially for a significant number of data pairs, and thus software is often used (such as Excel, Python, or R).

Develop a Linear Regression Model

In the following example, Python will be used for the following analysis based on a given dataset of (x, y) data.

- Create a scatterplot.
- Calculate the correlation coefficient.
- Construct the best-fit linear equation.
- Predict values of the dependent variable for given values of the independent variable.

There are various ways to accomplish these tasks in Python, but we will use several functions available within the `scipy` library, such as:

`Pearsonr()` for correlation coefficient

`linregress()` for linear regression model

`plt.scatter()` for scatterplot generation

Example 4.24

A marketing manager is interested in studying the correlation between the amount spent on advertising and revenue at a certain company. Twelve months of data were collected as shown in Table 4.16. (Dollar amounts are in thousands of dollars.)

Use Python to perform the following analysis:

- Generate a scatterplot.
- Calculate the correlation coefficient r .
- Determine if the correlation is significant (use level of significance of 0.05).
- Construct a linear regression model.
- Use the model to predict the revenue for a certain month where the amount spent on advertising is \$100.

Month	Advertising Spend	Revenue
Jan	49	12210
Feb	145	17590
Mar	57	13215
Apr	153	19200
May	92	14600
Jun	83	14100
Jul	117	17100
Aug	142	18400
Sep	69	14100
Oct	106	15500
Nov	109	16300
Dec	121	17020

Table 4.16 Revenue versus Advertising for a Company (in \$000s)

Answer

- The first step in the analysis is to generate a scatterplot. (Recall that scatterplots were also discussed in Scatterplots and Correlation.)

The Python function `plt.scatter()` can be used to generate the scatterplot for the (x, y) data. Note that we consider advertising to be the independent variable (x -data) and revenue to be the dependent variable (y -data) since revenue depends on the amount of advertising.

Here is the Python code to generate the scatterplot:

Python Code

```
# import the matplotlib library
import matplotlib.pyplot as plt

# define the x-data, which is amount spent on advertising
x = [49, 145, 57, 153, 92, 83, 117, 142, 69, 106, 109, 121]

# define the y-data, which is revenue
y = [12210, 17590, 13215, 19200, 14600, 14100, 17100, 18400, 14100, 15500, 16300, 17020]

# use the scatter function to generate a time series graph
```

```
plt.scatter(x, y)

# Add a title using the title function
plt.title("Revenue versus Advertising for a Company")

# Add labels to the x and y axes by using xlabel and ylabel functions
plt.xlabel("Advertising $000")
plt.ylabel ("Revenue $000")
```

The resulting output will look like this:

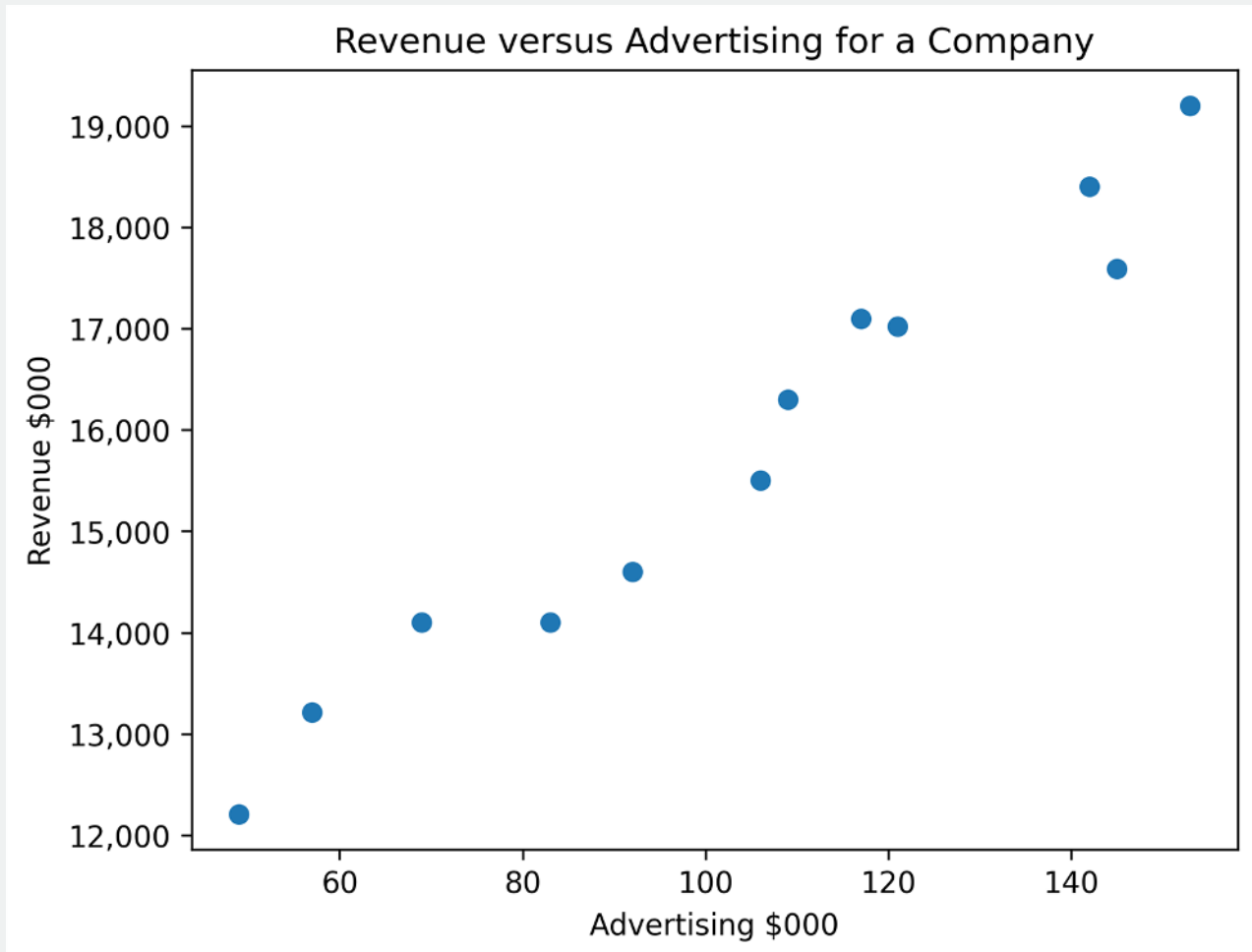


Figure 4.9 Best-Fit Line Superimposed on the Scatterplot

e. The regression equation is: $\hat{y} = 9,376.7 + 61.8x$.

The given amount of advertising spend of \$100 corresponds to a given x-value, so replace “x” with 100 in this regression equation to generate a predicted revenue:

$$\begin{aligned}\hat{y} &= 9376.7 + 61.8x \\ \hat{y} &= 9376.7 + 61.8(100) \\ \hat{y} &= 15556.7\end{aligned}$$

Since both advertising and revenue are expressed in thousands of dollars, the conclusion is that if advertising spend is \$100,000, the predicted revenue for the company will be \$15,556,700.

Interpret and Apply the Slope and y-Intercept

The slope of the line, b , describes how changes in the variables are related. It is important to interpret the slope of the line in the context of the situation represented by the data. You should be able to write a sentence interpreting the slope in plain English.

Interpretation of the Slope

The slope of the best-fit line tells us how the dependent variable (y) changes for every one-unit increase in the independent (x) variable, on average.

In the previous example, the linear regression model for the monthly amount spent on advertising and the monthly revenue for a company for 12 months was generated as follows:

$$\begin{aligned}\hat{y} &= a + bx \\ \hat{y} &= 9376.7 + 61.8x\end{aligned}$$

Since the slope was determined to be 61.8, the company can interpret this to mean that for every \$1,000 spent on advertising, on average, this will result in an increase in revenues of \$61,800.

The intercept of the regression equation is the corresponding y -value when $x = 0$.

Interpretation of the Intercept

The intercept of the best-fit line tells us the expected mean value of y in the case where the x -variable is equal to zero.

However, in many scenarios it may not make sense to have the x -variable equal zero, and in these cases, the intercept does not have any meaning in the context of the problem. In other examples, the x -value of zero is outside the range of the x -data that was collected. In this case, we should not assign any interpretation to the y -intercept.

In the previous example, the range of data collected for the x -variable was from \$49,000 to \$153,000 spent per month on advertising. Since this interval does not include an x -value of zero, we would not provide any interpretation for the intercept.

This page titled [1.6: Correlation and Linear Regression Analysis](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by [OpenStax](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.7: Key Terms

attribute

characteristic or feature that defines an item in a dataset

categorical data

data that is represented in different forms and do not indicate measurable quantities

cell

a block or rectangle on a table that is specified with a combination of a row number and a column number

comma-separated values (CSV)

format of a dataset in which each item takes up a single line and its values are separated by commas (“,”)

continuous data

data whose value is chosen from an infinite set of numbers

data

anything that we can analyze to compile some high-level insights

data analysis

the process of examining and interpreting raw data to uncover patterns, discover meaningful insights, and make informed decisions

data collection

the systematic process of gathering information on variables of interest

data preparation (data processing)

the second step within the data science cycle; converts the collected data into an optimal form for analysis

data reporting

the presentation of data in a way that will best convey the information learned from data analysis

data science

a field of study that investigates how to collect, manage, and analyze data in order to retrieve meaningful information from some seemingly arbitrary data

data science cycle

a process used when investigating data

data visualization

the graphical representation of data to point out the patterns and trends involving the use of visual elements such as charts, graphs, and maps

data warehousing

the process of storing and managing large volumes of data from various sources in a central location for easier access and analysis by businesses

DataFrame

a data type that

Pandas

uses to store a multi-column tabular data

dataset

a collection of related and organized information or data points grouped together for reference or analysis

discrete data

data that follows a specific precision

Excel

a spreadsheet program with a graphical user interface developed by Microsoft to help with the manipulation and analysis of data

Extensible Markup Language (XML)

format of a dataset with which uses tags

Google Colaboratory (Colab)

software for editing and running Jupyter Notebook files

Google Sheets

a spreadsheet program with a graphical user interface developed by Google to help with the manipulation and analysis of data

information

some high-level insights that are compiled from data

Internet of Things (IoT)

the network of multiple objects interacting with each other through the Internet

item

an element that makes up a dataset; also referred to as an *entry* and an *instance*

JavaScript Object Notation (JSON)

format of a dataset that follows the syntax of the JavaScript programming language

Jupyter Notebook

a web-based document that helps users run Python programs more interactively

nominal data

data whose values do not include any ordering notion

numeric data

data that are represented in numbers and indicate measurable quantities

ordinal data

data whose values include an ordering notion

Pandas

a Python library specialized for data manipulation and analysis

predictive analytics

statistical techniques, algorithms, and machine learning that analyze historical data and make predictions about future events, an approach often used in medicine to offer more accurate diagnosis and treatment

programming language

a formal language that consists of a set of instructions or commands used to communicate with a computer and instruct it to perform specific tasks

Python

a programming language that has extensive libraries and is commonly used for data analysis

qualitative data

non-numerical data that generally describe subjective attributes or characteristics and are analyzed using methods such as thematic analysis or content analysis

quantitative data

data that can be measured by specific quantities and amounts and are often analyzed using statistical methods

R

an open-source programming language that is specifically designed for statistical computing and graphics

recommendation system

a system that makes data-driven, personalized suggestions for users

sabermetrics

a statistical approach to sports team management

sports analytics

use of data and business analytics in sports

spreadsheet program

a software application consisting of electronic worksheets with rows and columns where data can be entered, manipulated, and calculated

structured data

dataset whose individual items have the same structure

unstructured data

dataset whose individual items have different structures

XML tag

any block of text that consists of a pair of angle brackets (< >) with some text inside

This page titled [1.7: Key Terms](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.8: Chapter Review

1.

Select the incorrect step and goal pair of the data science cycle.

- a. Data collection: collect the data so that you have something for analysis.
- b. Data preparation: have the collected data stored in a server as is so that you can start the analysis.
- c. Data analysis: analyze the prepared data to retrieve some meaningful insights.
- d. Data reporting: present the data in an effective way so that you can highlight the insights found from the analysis.

2.

Which of the following best describes the evolution of data management in the data science process?

- a. Initially, data was stored locally on individual computers, but with the advent of cloud-based systems, data is now stored on designated servers outside of local storage.
- b. Data management has remained static over time, with most data scientists continuing to store and process data locally on individual computers.
- c. The need for data management arose as a result of structured data becoming unmanageable, leading to the development of cloud-based systems for data storage.
- d. Data management systems have primarily focused on analysis rather than processing, resulting in the development of modern data warehousing solutions.

3.

Which of the following best exemplifies the interdisciplinary nature of data science in various fields?

- a. A historian traveling to Italy to study ancient manuscripts to uncover historical insights about the Roman Empire
- b. A mathematician solving complex equations to model physical phenomena
- c. A biologist analyzing a large dataset of genetic sequences to gain insights about the genetic basis of diseases
- d. A chemist synthesizing new compounds in a laboratory

This page titled [1.8: Chapter Review](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

1.9: References

Anaconda. (2020). *2020 state of data science*. <https://www.anaconda.com/resources/w...a-science-2020>

Clewell, A. (2024, January 26). *Three smart cities that failed within five years of launch*. Intelligent Build.tech. <https://www.intelligentbuild.tech/20...ars-of-launch/>

Hays, C. L. (2004, November 14). What Wal-Mart knows about customers' habits. *New York Times*. www.nytimes.com/2004/11/14/b...rs-habits.html

Herrington, D. (2023, July 31). *Amazon is delivering its largest selection of products to U.S. Prime members at the fastest speeds ever*. Amazon News. <https://www.aboutamazon.com/news/ope...delivery-speed>

Hitachi. (2023, February 22). *Ag Automation and Hitachi drive farming efficiency with sustainable digital solutions*. Hitachi Social Innovation. <https://social-innovation.hitachi/en...ty-in-farming/>

IABAC. (2023, September 20). *Fraud detection through data analytics: Identifying anomalies and patterns*. International Association of Business Analytics Certification. iabac.org/blog/fraud-detecti...s-and-patterns

Statista. (2024, May 10). *Walmart: weekly customer visits to stores worldwide FY2017-FY2024*. <https://www.statista.com/statistics/...res-worldwide/>

Van Bocxlaer, A. (2020, 20 August). *Sensors for a smart city*. RFID & Wireless IoT. www.rfid-wiot-search.com/rfi...r-a-smart-city

This page titled [1.9: References](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.

2: Units, Dimensions, and Conversions

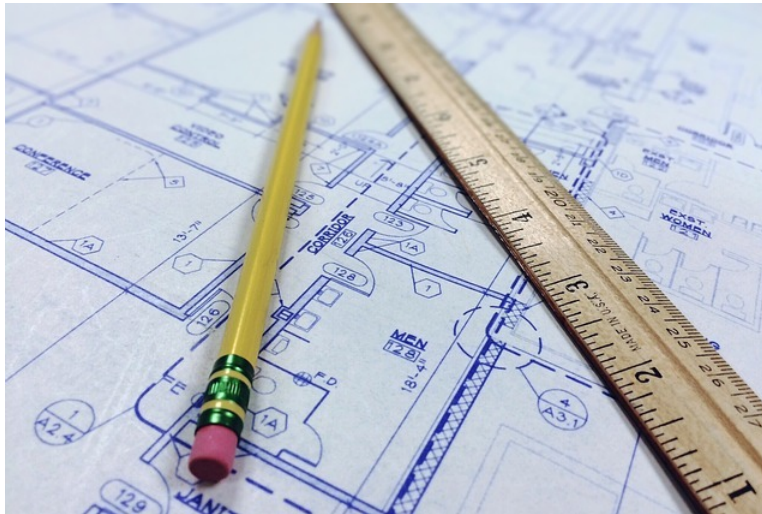


Figure 3.1: The important part is to understand when you need to convert units.

Although it seems basic, one of the first steps of becoming a successful engineer is having an **intuitive** understanding of units and dimensions. Intuitive means that when you see 4 *inches* or 4 *millimeters* you can “see” the difference between those two and understand how they are different. Intuitive means that when someone says that they are 6 meters tall, you raise your eyebrows and likely reply “I don’t think meters means what you think it means”. Unfortunately, it takes practice and time to have an intuitive understanding of anything (that is one of the reasons why we expect engineers to spend 4 years in school!). Therefore, the goal of this chapter is not that you will have an intuitive understanding by the time you are finished. It is important to understand that the learning process does not end at the completion of each chapter but instead is a lifelong process. The goal is that by the end of the chapter you will know how to appropriately look things up and that you understand how to be careful when converting units.

Learning Goals

In this chapter, we will learn a ton! We are going to learn the difference between units and dimensions and how you will be expected to manipulate them as an engineer. We will explore the following:

- The metric system.
- US Customary Unit system.
- How to convert between unit systems.
- The fundamental dimensions of the universe! (Grandiose but true!).
- Dimensional homogeneity (a fancy way of saying dimensional consistency) and analysis.
- How to intuit the difference between precision and accuracy.

Motivation - The Gimli Glider

On July 22, 1983, Air Canada flight 143 took off from Montreal to Edmonton. Before takeoff, routine checks showed all systems were functional and that everything was ready to go. While cruising at 12,500 *meters*, an alarm started blaring in the cockpit, warning the pilots that the fuel pressure on the left side of the airplane was low. The pilots silenced the alarm, figuring it was a fuel pump problem, and they knew gravity could handle moving the fuel around until they landed. The fuel gauges were not working correctly (or so it seemed), but the flight computer let the pilots know they had **plenty** of fuel to make the flight. A few minutes later, the right-side fuel pressure alarm went off. Seconds after that, the left engine shut down. The pilots called air traffic control and prepared to land with a single engine in Winnipeg. As they were discussing the plan with air traffic control in Winnipeg, the

“all engines down” alarm went off and the cockpit went blank. The pilots consulted the manual and couldn’t find a section that told them what to do when all engines fail. Luckily, the pilots managed to safely crash land the plane, and no one was hurt.



Figure 3.2: The Gimli Glider before it crashed...duh

Why did it crash?

I hope that you can guess based on the chapter title. The problem was an incorrect unit conversion (and a whole host of other errors)! In 1983 when the incident occurred, Canada was in the process of converting the country to the metric system. The new 767 involved in the incident was supposed to be calibrated using metric units, but all other Air Canada aircraft at the time were still operating with Imperial Units. The pilot calculated that the flight required 22,300 *kilograms* of fuel. A [float stick](#) check showed that there were already 7,682 *liters* of fuel in the tanks. Note that the calculation is in *kilograms*, a unit of mass, and the float stick indicated a volume, in *liters*. The crew used an incorrect *conversion factor* that related mass of a liter of fuel to pounds, an error that went unnoticed by the flight crew. We will explore the specific mathematical errors the crew made later in this chapter. The moral of the story, a flight crew made a bad conversion, that crashed a plane. Luckily, no one died but the story highlights the importance of converting units carefully.

Units vs Dimensions

It is easy to get units and dimensions mixed up. It helps to think of dimensions as **fundamental and usually unspoken** whereas units are **derived from the dimension and quantify the dimension**.



Figure 3.3: In basketball, taller is usually better.

As an example, let's say you are recruiting someone to play on your basketball team. You know that in basketball, there are significant advantages for tall players. You might ask a potential Canadian player "How tall are you?" and she might reply "I am 1.8 *meters* tall". In this case, the **dimension** we are considering is *length* (aka "height" in colloquial terms), and the **unit** we are using to describe the dimension length is **meters**. However, if you ask an American player who is the exact same height as the Canadian player, she might reply "I am 5 *feet* 11 *inches* tall". In this case, the **dimension is still** *length*, but now the units being used to describe that dimension, are now **feet and inches**.

Dimensions

To summarize what you read in the previous section, a **dimension** describes the measurement of interest, which in the case above was *length*. There are no other dimensions necessary nor does it even make sense to think about another "dimension to height". On the other hand, the **unit** *meter* is used to quantify the dimension and there are multiple different units we can use. Some units are more practical than others as we will see later in this chapter. For now, let's concentrate our mental effort on understanding dimensions.

The Fundamental Dimensions of the Universe

Now that we have an idea of what a dimension is, we can easily list out **all of the dimensions needed to describe anything in the universe**. Yes, that is a bold statement but it is true. It turns out there are only a few fundamental dimensions of the universe and we can measure every single physical phenomenon you can think of using combinations of these dimensions.

Dimension	Symbol
Mass	M
Length	L
Time	T
Temperature	Θ
Amount of Substance	N
Light Intensity	J
Electric Intensity	I

Table 3.1: The fundamental dimensions of the universe and their corresponding symbols.

Table 3.1 shows the seven fundamental dimensions and the symbol used to define them. That is it. That is all the dimensions you need to describe everything in the universe.

Hold on a second, you might say. What about something like *volume*? Isn't *volume* a dimension? Well yes, but it is a **derived dimension**. The dimensions listed above are the fundamental dimensions that we can then combine in different ways to arrive at derived dimensions.

Derived Dimensions

You are already familiar with some of the derived dimensions. For example, it is likely that you already understand that *volume* is really just $length * length * length = length^3$. Similarly, you probably already knew that the dimension *area* is simply $length * length = length^2$. As we explore more complex concepts in engineering, you can expect the dimensions to get more complex.

A Quick Note on Notation

Now is a good time to put this concept into practice. Before we do, just a quick note on notation. Consider the dimension of *velocity*. You may have learned in physics class (or will learn) that $velocity = \frac{distance}{time}$. Now, we remember that distance is really the same thing as the dimension *length* which is represented by the symbol L . Similarly, Table 3.1 shows that time is represented by the symbol T . Therefore, the dimensions of velocity is $\frac{L}{T} = LT^{-1}$

As you can see, all of the ways of writing out the notation are equally as valid. However, the LT^{-1} is the most space efficient and the form that we will be using for the rest of the chapter. [If you need a refresher on the math behind this, click here.](#)

Other derived dimensions would include Force, Pressure, density, and acceleration. Use these examples of each physical properties and units to help you answer Question 3.2.

Examples: The density of water is $1 \frac{g}{ml}$. Atmospheric pressure is 101,325 Pa where Pa is a unit know as Pascal which is equal to $1 N/m^2$. Gravitational Acceleration is $9.81 m/s^2$. That's all the hints for now!

The dimensions are written in the "negative exponent" notation to indicate division for the sake of saving space.

Deep Dive Into the Dimensions of Something a Bit More Complex

Now that we have covered some of the simple quantities and your brain muscles are loose, let's take one more look at dimensions and consider the physical concept of *work*. Take a minute to watch the following YouTube clip to familiarize yourself with the concept of Work.



Video 1: Video Courtesy of PBS Digital Studios: Work, Energy, and Power

So what are the dimensions of *work*? Let's logically follow the steps below:

Step 1) From the video we saw that $Work = Force * Distance$. We already know that *distance* has dimension of *length*, but we don't have any idea about the dimensions of *force*. That leaves us here.

$$"Dimension of Force?" * L \quad (2.1)$$

Step 2) Well, luckily for us, the video also states that $Force = Mass * Acceleration$ and we know that *mass* is a fundamental dimension. So we can then expand our analysis.

$$M * "Dimension of Acceleration?" * L \quad (2.2)$$

Step 3) We are making progress but now we have another unknown dimension, the dimensions of *acceleration*. But wait! You should have found above in Question 2 that *acceleration* has dimensions of LT^{-2} . Now we just need to put it all together!

$$M * L * T^{-2} * L = M * L^2 * T^{-2} \quad (2.3)$$

That means our final answer is: ML^2T^{-2}

Work can be defined as the transfer of energy, so it turns out the Work and Energy share the same dimensions. That means you also now know the dimensions for Energy as well. Now try and do this on your own by filling in the blanks to Question 3.3 where electric charge has dimensions of electric intensity * time and electric potential is defined as the work done to move a unit of charge.

I tricked you into learning something, surprise! You have just done **dimensional analysis**! Dimensional analysis is a powerful tool in engineering. It allows us to understand a physical quantity by understanding the underlying fundamental dimensions of the quantity. In fact, if you go to the Wikipedia page for [work](#) you will see they list the dimensions of work as part of the summary.

Copy and Paste
Image here.
Delete this
placeholder image

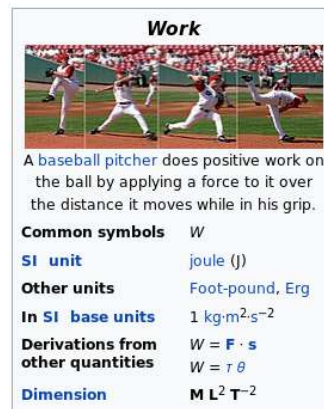


Figure 3.5: Screenshot of Wikipedia page for Work showing the dimensions. I told you it's important.

Unit Systems and History

Take another look at figure 3.5 above. Notice how it talks about the “SI unit” used to describe work? You can also see that there is a little section describing “Other units”. Wikipedia is following the scientific and international standard of using the SI System. So what is the SI system and why did Canada risk crashing an airplane to switch?

The History of the SI System (aka the Metric System)

SI stands for The International System of Units. It is derived from the french, French *Le Système International d'Unités* (hence the seemingly back words abbreviation, SI). The SI system is commonly referred to as the metric system and both metric and SI will be used interchangeably throughout the book. The history of SI is pretty interesting. Watch the following video to get a feel for how SI units came to be.



Video 2: A nice TEDEd video that describes the history of units. Easily worth a watch! You could use a quick little break anyway...

Let's say that you get elected President of the United States and you make the decision that the US should switch to SI. You make an executive order, and overnight, the US is supposed to switch to the metric system. For this discussion, think of one good consequence, and one potentially disastrous consequence from America switching to metric overnight.

What Engineers Need to Know About the SI System

There are three main concepts that you need to fully understand the SI system: The Base Units, Derived Units, and Metric Prefixes. Once you understand these three concepts you are well on your way to being an SI expert! *Do you notice the similarities of "The Base Units" and "Derived Units" to our sections on dimensions?*

The Base Units

The SI base units are the building blocks of the whole system. They are analogous to the fundamental dimensions discussed in this chapter earlier. Since the units are simply descriptors of the quantity of a dimension, let's go ahead and pair them up!

Dimension	Symbol	SI Unit Matched to Dimension	SI Unit Symbol
Mass	M	kilogram	kg
Length	L	meter	m
Time	T	second	s
Temperature	Θ	kelvin	K
Amount of Substance	N	mole	mol
Light Intensity	J	candela	cd
Electric Intensity	I	ampere	A

Table 3.2: The fundamental dimensions matched to the SI unit.

So when we want to talk about the height of a basketball player, a scientist (or as we learned from the video, anyone not in the US, Myanmar, or Liberia) would say that Nikola Jokic is 2.11 *meters* tall. The dimension is *length* and the SI unit for *length* is the meter. Similarly, in the SI system Michael Jordan is listed as having a *mass* of 98kg. The dimension, in this case, is *mass*, and the associated SI unit is the kilogram.

The only common exception to this rule is for the dimension of *time*. If you ask a scientist how old Nikola Jokic is it is highly unlikely that they would reply that he is 1767226000 *seconds* old unless they were *really* into the SI system. However, for any of the other dimensions listed above, you should use the SI unit for the dimension when you can.

Derived Units

Just like the dimensions, we can then combine different combinations of units to help us quantify more complex physical dimensions.

For example, we discussed the concept of force above. We already determined that the dimensions of force are MLT^{-2} . Similarly, force has a **derived SI unit** of Newtons. Remember, unlike dimensions, units are quantities that describe dimensions, and the values attributed to them are arbitrary.

What I mean by arbitrary is let's consider a different derived unit, the *Joule*. 1 *Joule* is defined as the amount of work done moving 1 *Newton* by 1 *meter*. That is arbitrary. Why isn't 1 *Joule* defined as moving 10 *Newtons* over 12 *meters*? Although it seems easier to use the number 1 (it is easier) it doesn't HAVE to be that way. It just is because that's what a bunch of french dudes decided. That being said, they did make some great decisions that make calculations and understanding units much easier as we will see.

Remember: **Dimensions are universal concepts but units are not.** What I mean by that is if we ever meet aliens, they will know what the dimension of *length* is and what it refers to, but they will have no idea what a meter is.



Figure 3.7: If we ever meet aliens they WILL know about length, but they WON'T know what a meter is.

Some of the derived units are pretty easy. Consider *area*. Since the base SI unit for *length* is the meter, and we know that $area = length * length$, it follows that the SI unit for area is simply $meter * meter = meters^2$.

There are currently 22 derived units with special names. [Click here to see the list](#). You will notice that almost all of the names are the names of famous scientists and engineers who were involved in the discovery of the scientific concept covered by the unit. A table of the most common derived SI units is shown below.

Dimension (Symbol)	SI Unit	Base SI Units	Derivation Equation
Force (F)	Newton	$kg\ m\ s^{-2}$	$F = m * a$
Work (W)	Joule	$kg\ m^2\ s^{-2}$	$W = F * x$
Power (P)	Watt	$kg\ m^2\ s^{-3}$	$P = E / t$
Pressure (p)	Pascal	$kg\ m^{-1}\ s^{-2}$	$p = F / A$

Table 3.3: A small collection of the derived units.

Metric Prefixes

Up until this point, it may seem there is little to no difference between the metric system and USCU (United States Customary Units) and you are right! USCU also has a unit for *length* (e.g. the *inch*) and you can also create derived units for *area* (e.g. the *inch*²). If you live in the United States or are familiar with the USCU you should know that there are actually **multiple different units used for length**. In the US they are most commonly, the *inch*, the *foot*, the *yard*, and the *mile*. You should also know that the *mile* is used for long distances, the *yard* for intermediate distances, etc. The different units are designed to allow us to talk about different quantities easily. But converting between *inches* and *yards* is NOT straightforward.

The beauty of the metric system is that it does not have an equivalent confusing structure. **There is ONLY the meter for the dimension of length**. So what happens when we want to talk about different magnitudes of different quantities? That is where the metric prefixes come into play. **The prefixes work by changing the magnitude of the dimension**. For example, if we are talking about the height of a person, it is likely that it will be described in numbers of meters. However, if we are talking about the circumference of the earth, we would likely talk about **kilometers**. In this example, the metric prefix is “kilo” which stands for 1000. So instead of saying the circumference of the earth is 40,000,000 meters we would say that it is 40,000 kilometers. Table 3.4 shows the *most common* metric prefixes:

Text	Symbol	Power
tera	T	10^{12}
giga	G	10^9
mega	M	10^6
kilo	k	10^3
---	---	---
deci	d	10^{-1}
centi	c	10^{-2}
milli	m	10^{-3}
micro	μ	10^{-6}
nano	n	10^{-9}

Table 3.4: A small collection of SI prefixes.

Keep in mind that there are more metric prefixes than this. For a complete list, [check out this page](#).

Now we can combine any of the metric prefixes with any of the units to change dimension. For example, let's consider a blink of an eye.



Figure 3.10: In the blink of an eye...

It takes about 0.3 to 0.4 seconds to blink your eye. We could say that it takes 3 to 4 deciseconds because the prefix “deci” corresponds to 10^{-1} .

$$0.3 \text{ seconds} = 3 \text{ deciseconds} = 3 \times 10^{-1} \text{ seconds} \quad (2.4)$$

However, no one really talks about deciseconds, so instead, you might say that an eye blink takes 300 to 400 **milliseconds**. In this case the prefix “milli” corresponds to 10^{-3} so it is still equivalent (see table 3.4).

Make sure to work out your brain until this concept makes sense to you! You can make up your own problems easily here depending on what you are into. Do you like swimming? How many **centimeters** are there in a 100 *meter* swim? You get the idea.

Metric Prefixes and Derived Units

One thing to keep in mind when using derived units is that the metric prefix used to define the unit in Table 3.3 matters. You will note that for example Newton is defined as $1 \frac{\text{kg} \cdot \text{m}}{\text{s}^2}$. Therefore, if you were solving for the Force and were given units of mass as 10 g and acceleration as $13 \frac{\text{cm}}{\text{s}^2}$ then you would need to convert g to kg and cm to m in order to get a Newton. This is shown in the example below:

$$F = m \cdot a = 10 \text{ g} \cdot 13 \frac{\text{cm}}{\text{s}^2} = 130 \text{ g} \frac{\text{cm}}{\text{s}^2}$$

$$130 \frac{\text{cm} \cdot \text{g}}{\text{s}^2} \cdot \frac{(1 \text{ kg})}{1000 \text{ g}} \cdot \frac{(1 \text{ m})}{100 \text{ cm}} = .0013 \text{ kg} \frac{\text{m}}{\text{s}^2} = .0013 \text{ N}$$

The United Customary System Unit System

The United States Customary System (USCS) units are the system of measurements used in the United States of America. If you have grown up in the United States you may have better intuition on already when it comes to USCS units. For example, if I told you that I was 5 ft 3 in you'd be able to quickly understand that versus if I told you I was 1.6 meters. Same thing if I told you to buy 1 gallon of milk versus 3.8L of milk. However, they can be more difficult to use in calculations when it comes to remembering conversion factors between units within the same dimension as these values are much more arbitrary than the metric prefix. If you would like to read more about USCS units you can find the [wikipedia page here](#).

The dimensions, SI unit, and USCS unit are provided in the table below. One important thing to note when it comes to USCS units compared to SI units is that mass is a derived unit as opposed to force. Therefore, you will find in many engineering problems that when you are in SI units, you will be provided with a mass of the object. However, if you are using USCS units, you will be provided the weight of an object. These are different dimensions and therefore cannot be converted the same way you can convert meters to feet which are both length dimensions. We'll talk more about this later in the chapter.

Dimension	SI Unit	USCS Unit
Mass	kilogram (kg)	slug = (lbs ² /ft)
Force	Newton (N) = (kgm)/s ²	pound (lb)
Length	meter (m)	foot (ft)
Time	second (s)	second (s)
Temperature	Kelvin (K)	Fahrenheit (F)

Table 3.5 USCS Units

Changing Magnitude of the Dimension

As shown in Table 3.4, SI units have prefixes that are based on a power of 10 to change the magnitude of the dimension. USCS units, however, are not as simple to change from one magnitude to another and require a unit change such as feet to inches or feet to miles. Some of these unit changes are provided below:

$$1 \text{ foot} = 12 \text{ inches} \quad (2.5)$$

$$1 \text{ mile} = 5280 \text{ feet} \quad (2.6)$$

$$1 \text{ ton} = 2000 \text{ lb} \quad (2.7)$$

$$1 \text{ kip} = 1000 \text{ lb} \quad (2.8)$$

It is important to note that both ft and in are units for dimensions of length which is why you can convert between them. The same goes for the units ton, lb, and kip as these are all units for the dimension of force.

Let's do an example of unit conversion between feet and inches. My nephew is 77 inches tall. I think for most of us we wouldn't be able to intuitively know if that was tall or short without converting to feet for a better understanding. So let's do that:

$$77 \text{ in} * 1 \text{ ft}/12 \text{ in} = 77/12 \text{ ft} = 6.42 \text{ ft}$$

6.42 ft sounds pretty tall! We are used to seeing height in units of both ft and inches so now let's try to turn that .42 into inches:

$$.42 \text{ ft} * 12 \text{ in}/1 \text{ ft} = 5 \text{ in}$$

We also use unit conversions to switch between unit systems. Let's talk more about unit conversions in the next chapter.

Converting Units and Conversion Factors

Up until this point, you should have a good grasp of what a dimension is, and what a unit is. We have also discussed a few reasons for unit conversions such as changing magnitude or unit, obtaining the units needed for a derived unit, and another reason may be because we need to switch between the unit systems. Now the question becomes, how can we use all of this knowledge to convert between different units and prefixes? The last piece of the puzzle is going to be conversion factors. If you can't find the conversion factor you need, google it! We will use conversion factors as a kind of "decoder ring" to help us arrive at the unit of interest. To start, let's begin with some of the conversion factors for the dimension of Length.

Various Conversion Factors for Length
1 meter = 3.281 feet
1 kilometer = 0.621 miles
1 foot = 12 inches
1 inch = 2.54 centimeters
1 mile = 5280 feet

Table 3.6: Some conversion factors for length.

Various Conversion Factors for Mass and Force
1 slug = 14.49 kg
1 lb = 4.45 N

Table 3.7: Some conversion factors for mass and force.

There are a couple of things to notice from the table above. First, the conversion factors can be between USCS and metric (e.g. 1 *meter* = 3.281 *feet*) or from USCS to USCS (e.g. 1 *foot* = 12 *inches*). The reason you don't see any conversion factors from metric to metric is because we use *prefixes* instead of different units to describe quantities of different magnitudes.

The way to read the conversion factors is "There are 0.621 miles for every 1 kilometer" or mathematically that can be represented as:

$$1 = \frac{0.621mi}{1km}$$

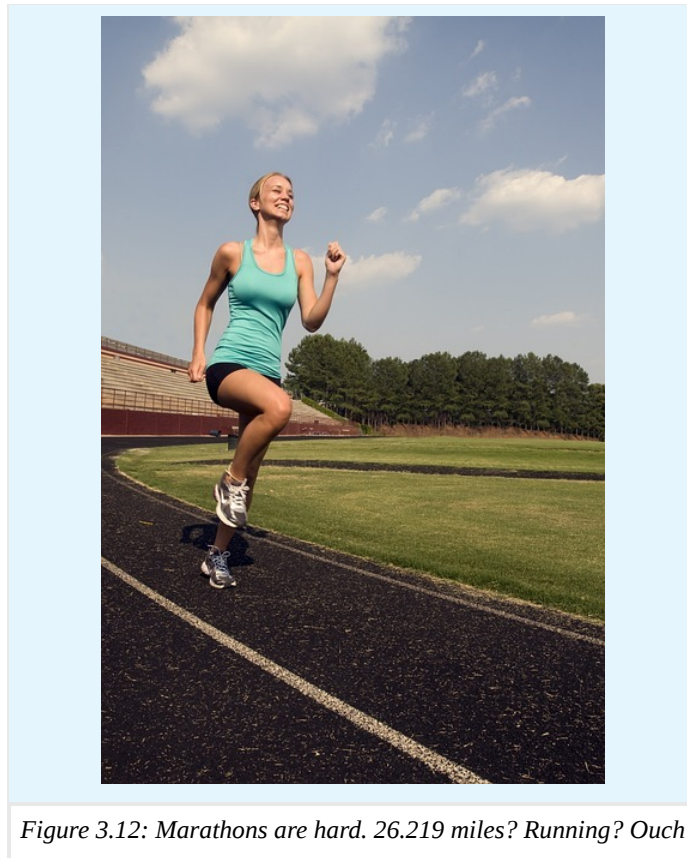
It is important to note that since $0.621\ mi = 1\ km$ that the ratio $\frac{0.621mi}{1km}$ is actually a dimensionless 1. Now, time for a quick math problem...

OK, well that was an easy question but the idea is to get you to consider that the ratio of $\frac{0.621mi}{1km}$ is the same as $\frac{1km}{0.621mi}$ which is the same as the number 1. We also know that we can multiply any number by 1 without changing the number. OK, so why is this useful? Let's look at an example to illustrate.

Marathon Example (Converting Length)

Let's say that we wanted to convert the number of miles in a marathon to kilometers. We know that a marathon is 26.219 *miles*. We know that our conversion factor tells us that $0.621\ mi = 1\ km$. Now all we have to do is multiply our original quantity, 26.219 *miles*

by one of the two ratios: $\frac{0.621mi}{1km}$ or $\frac{1km}{0.621mi}$. But which one should we use? **The trick is to select the ratio such that the unit that is being converted is eliminated, and the unit we are trying to convert to remains.**



If we try the first ratio we get: $26.219mi * \frac{0.621mi}{1km} = 16.282 \frac{mi^2}{km}$ which is clearly a completely meaningless answer. What are the units $\frac{mi^2}{km}$ and what does that have to do with distance? We acknowledge that *technically* that answer is correct because the quantity $\frac{0.621mi}{1km}$ is just the number 1, *but it does not answer our original question.*

However, if we try the second ratio we get: $26.219mi * \frac{1km}{0.621mi} = 42.221km$ we can see that the unit *mi* cancels out algebraically and we are left with *km* for our answer. So therefore we can say that a marathon is 26.219 *mi* or 42.221 *km*.

Putting this all together we can come up with...

Steps for converting units:

1. Determine the appropriate conversion factor(s) to be used.
2. Set up a ratio based on the conversion factor that will result in the current unit being eliminated, and the unit of interest being preserved.
3. Perform multiplication and algebra.

Complicated Conversions - Units as Ratios

In the previous example (and a lot of the time in engineering applications), conversions can be completed in one step and with one conversion factor. Sometimes, instances will arise where you need to perform multiple steps and use multiple conversion factors. As long as you understand how to use the steps for converting units, you can apply that logic to more complicated situations.

For example, let's consider converting $\frac{55 \text{ miles}}{\text{hour}}$ to $\frac{\text{kilometers}}{\text{second}}$. In this case, we can see that we are going to need to convert *miles* to *kilometers* and *hours* to *seconds*. Although this seems more complicated, it is almost the exact same conversion process as we have done before. This time though, we need to do 2 conversions. See if you can follow along with the steps we outlined above:

Step 1) Determine the appropriate conversion factors to be used

In this case, the conversion factors of interest will be:

$$1 \text{ kilometer} = 0.621 \text{ miles AND } 1 \text{ hour} = 3600 \text{ seconds}$$

Step 2) Set up a ratio based on the conversion factor that will result in the current unit being eliminated, and the unit of interest being preserved.

$$\frac{1 \text{ kilometer}}{0.621 \text{ miles}} \text{ AND } \frac{1 \text{ hour}}{3600 \text{ seconds}}$$

We set up the ratios this way so that when we multiply $55 \frac{\text{miles}}{\text{hour}}$ by those ratios, we can see that miles will be eliminated and kilometers will remain in the numerator AND hour will be eliminated but seconds will remain in the denominator.

Step 3) Perform multiplication and algebra

Multiplying everything and canceling reveals:

$$55 \frac{\text{miles}}{\text{hour}} * \frac{1 \text{ kilometer}}{0.621 \text{ miles}} * \frac{1 \text{ hour}}{3600 \text{ seconds}} = 0.0246 \frac{\text{kilometers}}{\text{seconds}}$$

Complicated Conversions - Units with Exponents

The last thing you need to be aware of when you are performing unit conversions is when we look at squared, cubed, and other higher power units. For example, let's consider the dimension *volume*. We know that SI unit for the dimension *volume* is the cubic meter (m^3). The important thing here is the “cubic” part. If we want to convert from cubic meters to say, cubic inches, we can use the conversion factors: $1 \text{ meter} = 3.281 \text{ feet}$ and $1 \text{ foot} = 12 \text{ inches}$ in a two-part process to eliminate meters and end up with inches. Notice, however, that our conversion factors are NOT in powers of three. We have to account for that when we are performing our conversions. All you have to do, **is cube the ratios that you develop** during Step 2 of our conversion process technique.

To illustrate this, let's convert $12.3m^3$ to in^3 . We will follow the same steps we used in the previous examples.

Step 1) Determine the appropriate conversion factors to be used

In this case, we already discussed that the conversion factors would be:

$$1 \text{ meter} = 3.281 \text{ feet and } 1 \text{ foot} = 12 \text{ inches}$$

Step 2) Set up a ratio based on the conversion factor that will result in the current unit being eliminated, and the unit of interest being preserved.

This is where things get tricky. In order for this to work, we can tell that we need cubic feet in the numerator for the first multiplication step and cubic meters in the denominator. However, our conversion factors identified do not correspond to the cubic quantities. **The solution is simply to cube the appropriate ratio to create a new conversion factor.** For this example the two conversion ratios then become:

$$\frac{3.281 \text{ ft}}{1 \text{ m}} * \frac{3.281 \text{ ft}}{1 \text{ m}} * \frac{3.281 \text{ ft}}{1 \text{ m}} = 35.32 \frac{\text{ft}^3}{1 \text{ m}^3}$$

AND

$$\frac{12 \text{ in}}{1 \text{ ft}} * \frac{12 \text{ in}}{1 \text{ ft}} * \frac{12 \text{ in}}{1 \text{ ft}} = \frac{1728 \text{ in}^3}{1 \text{ ft}^3}$$

Step 3) Perform multiplication and algebra

Now that we have conversion factors with appropriate units, we can perform the multiplication steps and eliminate unwanted units.

$$12.3m^3 * \frac{35.32 \text{ ft}^3}{1 \text{ m}^3} * \frac{1728 \text{ in}^3}{1 \text{ ft}^3} = 750705 \text{ in}^3$$

Warning on Unit Conversions

Now that we have seen lots of unit conversion examples, I want to remind you that you can only convert units of the same dimension. So you can convert meters to feet or feet to centimeters because those are all units of Length. You can convert Newton to pound because those are both units of force. Anytime you are provided with information in one dimension and need an answer in a different dimension, you will need to have an equation that relates those dimensions in order to solve.

One example of this is kg to lb. Most people in the United States will give their weight in units of lb while those who use SI system will often give you theirs in units of kg. Remember that kg is a unit for the dimension of mass while lb is a unit for the dimension of force. The equation used to relate mass to force is $F = m * a$. Therefore, if you want to find out how many kgs are in 230 lbs, you will need to convert 230 lb to a mass and then convert to kg. Let's see what this looks like.

$$F = ma$$

$$230 \text{ lb} = m * 32.2 \text{ ft/s}^2$$

$$m = 230 \text{ lb} / (32.2 \text{ ft/s}^2) = 7.14 \text{ s}^2/\text{ft} = 7.14 \text{ slug}$$

Now that we have a mass, we can convert slug to kg.

$$7.14 \text{ slug} * 14.59 \text{ kg/slug} = 111 \text{ kg}$$

Ethics

Humans are fantastically complicated and intelligent (sometimes) creatures but we still make mistakes on the occasion. Sometimes the consequences are small (i.e. a typo in a textbook) or sometimes the consequences are huge (i.e. a plane crashes and hundreds of people die).

In engineering, making mistakes can have *huge* consequences. You can find examples of engineering mistakes killing people all over the place. For example, the [Colombia space shuttle disaster](#) resulted from a piece of foam that came off of a tank and cut through the wing. An engineer designed [a crappy bridge at FIU](#) and it collapsed killing people. The list goes on, and the point is, that when engineers make mistakes, disasters happen.

The ethics questions I would like you to consider for this chapter is: who is to blame and how should we punish them?

This page titled [2: Units, Dimensions, and Conversions](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 03: Units, Dimensions, and Conversions](#) is licensed [CC BY-NC 4.0](#).

CHAPTER OVERVIEW

3: Introduction to programming and logic

[3.1: Thinking Algorithmically](#)

[3.2: Relational and Logical Operators](#)

[3.3: Documentation and such](#)

[3.4: Programming in any language](#)

[3.5: Mathematical Models](#)

This page titled [3: Introduction to programming and logic](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

3.1: Thinking Algorithmically

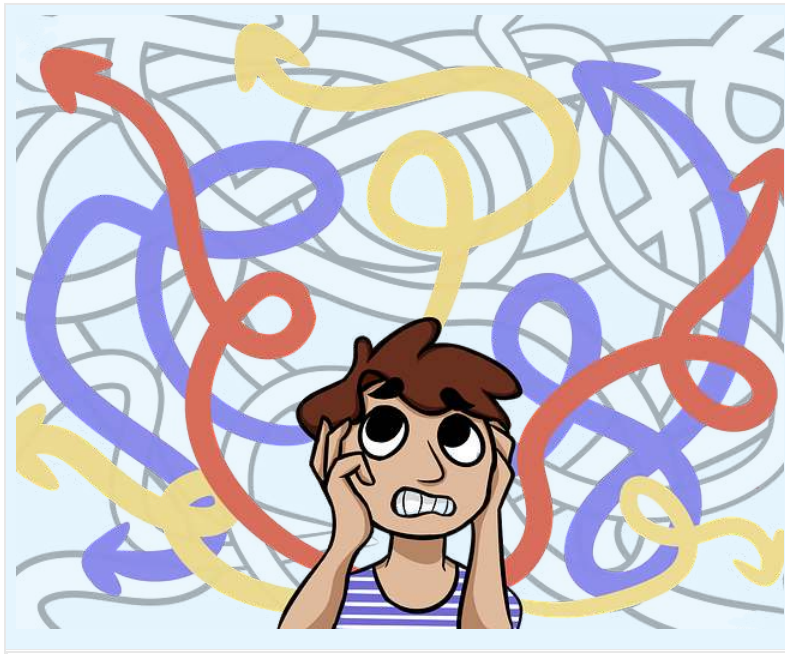


Figure 12.1: Think algorithmic thoughts.

Professional engineers are well known for their ability to solve complex problems. Throughout the rest of this book, we will continue to learn how to use MATLAB as a tool to solve problems. However, before we jump back into MATLAB syntax and usage, it is important to remember the end goal: **to learn how to solve problems algorithmically**.

In the spirit of learning how to think algorithmically, in this chapter you will be presented with material that will help you start to *think* like a computer. The key to it all is to remember that computers are really, really, really, dumb.

Learning Goals

In this chapter, we will take a break from MATLAB to learn how to think algorithmically. That means we will explore the following:

- How to approach solutions to problems
- What an algorithm is
- Where algorithms arise in everyday life and in engineering contexts
- How to “think algorithmically”
- How to write pseudocode
- How to create flowcharts

The 8 AM Attendance Problem

Getting to a class that is held early in the morning is a problem that some college freshmen and professors can relate to. Although, we professors typically complain about it much less (*and some of us, myself included, even enjoy early classes!*). Based on the attendance of classes I have taught at 8 AM, it **really is a problem that doesn’t have a definite solution yet!** There are a ton of

things that you need to do, to solve the problem of attending class at 8 AM. First, you have to start by waking up at a specific time that is often earlier than the time you are accustomed to waking up at. Presumably, your solution to this problem will look similar to my solution. Certain details may be different but you get the idea.

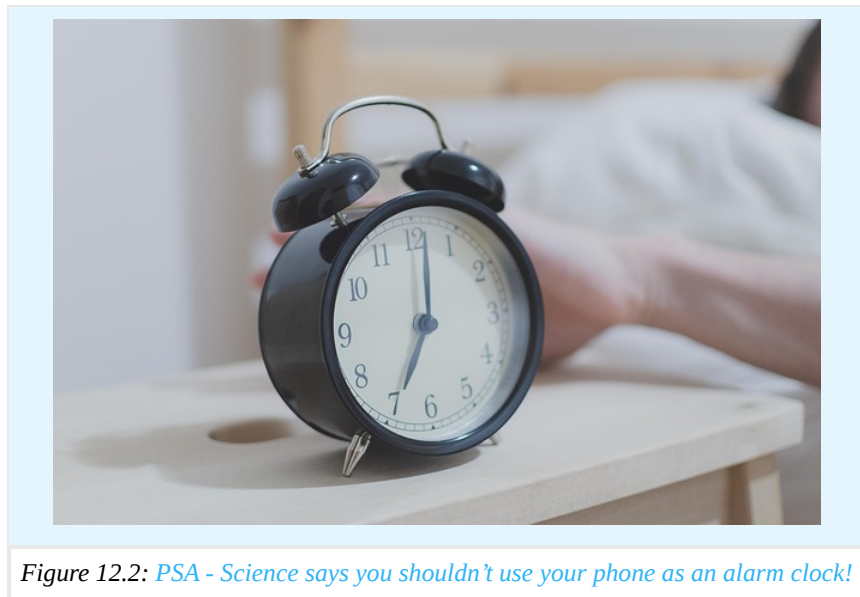


Figure 12.2: *PSA - Science says you shouldn't use your phone as an alarm clock!*

Professor's Solution - Speech

I wake up using a cheap digital alarm clock I have on my bedside table set for 6 AM. It cost me about \$4 and as a bonus, I avoid the temptation to check my smartphone before I go to bed. I found that I sleep better that way. I am not really a “snooze” guy, so I just pop up out of bed as soon as the alarm goes off. I take a shower, appreciating the clean, reliable, hot water, being delivered to my wall. If only my ancestors could imagine such luxury, such magnificence! Next, I check the weather on the internet, then put on some weather-appropriate clothing for bike riding. I hop on my bike and ride towards Colorado State University. I am fortunate that my commuting route doubles as a scenic route. It travels through an area of Fort Collins known for its beautiful turn-of-the-century houses and great, big, old, happy, trees. As I pedal through Old Town, I remind myself how lucky I am to live in a city with drivers that are educated on bicycle safety and are respectful to riders. I arrive at CSU, lock my bike, and get to class at 7:50 am.

What is your solution?

Congratulations you just wrote out an algorithm! You just authored a list of step-by-step instructions to solve a problem. That is what an algorithm is!

An algorithm is a set of instructions for solving a problem, step-by-step.

Millions of algorithms are present in our everyday life, it is just not common for us to acknowledge them as such. When we bake, we follow recipes that are just baking algorithms. That one is obvious but there are many more examples! When you are buying groceries you automatically gravitate towards the checkout line that *appears* as if it will take the least amount of time. Think about how complicated *that* algorithm is. You would likely prefer to be behind two people with three items each than one person with 50

items. Most of the algorithms that humans follow are just outside of our conscious thought so we don't realize that they are there even though our human algorithms govern almost every decision we make and every action we take.

How to write algorithms people (or computers) can actually follow

Part of learning how to be a professional engineer is learning how to do things efficiently, in a way that is repeatable by others. It is important that if you solve a unique problem, that others don't need to solve it all over again.

Now, if I wanted to describe to someone my solution to the 8 AM attendance problem, I could write it out as I did above (see Professor's Solution). However, there are some language flourishes that are unnecessary, take up space, and *do not contribute to your understanding of how I arrive at the solution*. If we strip those away, it then becomes clear that there are a lot of details that are missing. For example, if you wanted to copy my solution exactly, you may wonder what types of clothes I wear and when I wear them. Also, you may wonder if I ride my bike to work in all weather conditions. Do I ever drive? Etc.

For your algorithms to be usable by other people, **the algorithm needs to be as specific as possible**. You need to write algorithms down in such a way that a computer (sometimes, but less often, a human) can follow the steps to complete the solution. It is important to write your algorithms so that *nothing is implied*! To make sure I am writing my algorithms appropriately, I pretend that I am writing them for a really, really, dumb person. Even though we say things with computers are "smart" (i.e. smartphones), in reality, computers are shockingly dumb. They are just really *fast* which we can confuse as smart. For now, when crafting algorithms, remember to be as specific as you possibly can.

It is important to write your algorithms so that nothing is implied!

Let's break down the Professor's Solution to the problem as an ordered list of instructions that should be followed step-by-step:

Professor's solution - Algorithm

Below, is the Professor's Solution to the 8 AM attendance problem.

Instructions to ensure success when following the algorithm that solves the 8 AM attendance problem:

- Steps should be read from the top to bottom (line numbers are displayed in yellow to the left)
- Follow the instructions on each line to their logical conclusion
- Either *continue one line down* OR *follow the instruction on line designating skip location*

```
1 Wake up from alarm at 6am
2 Take a shower
3 Check the weather to get the forecast
4 If forecasted temperature >= 70 continue, else go to line 7
5     Wear t-shirt and shorts
6     Skip to line 17
7 If forecast >= 50 continue, else go to line 10
8     Wear long-sleeve shirt and jeans
9     Skip to line 17
10 If forecast >= 30 continue, else go to line 13
11     Wear jacket and jeans
12     Skip to line 17
13 Else
14     Wear sweatshirt and slacks
15     Leave house at 7:30am, drive car to CSU
16     Skip to line 19
17 Leave house at 7:20am to ride bike to CSU
18 Lock bike safely
19 Arrive at class by 7:50am
```

Figure 12.3: Professor's solution to the 8 AM problem.

Let's take a look at line 4: `If forecast >= 70 continue, else go to line 7`

Translated from Algorithm-ese into English, that is equivalent to saying:

If the temperature is **forecast** to be **greater than or equal to** 70 degrees F, **continue** to line 8 which tells you to wear a t-shirt and shorts. Or **else** you should consider other clothing options.

Let's say the forecast indicates it will be 61 degrees today. Therefore when I evaluate my algorithm, the statement `If forecast >= 70 continue,` evaluates to false. It isn't true, so I shouldn't follow the instruction. This is where the algorithm following rule **Follow the instructions on each line to their logical conclusion** comes into play. Since it is not true that it is 61 degrees, I should follow the else go to line 7 bit. So I skip to line 7 and am on my merry way.

Bugs

Notice how the steps are written in such a way, **that you can not mess them up if you follow the instructions**. If you start from the top and follow the instructions, you can't both ride the bike AND drive the car at the same time. You will not be instructed to wear both a t-shirt and a long sleeve, etc. You may think the way of writing `go to line 7` is unnecessary because you are smart and can understand that some things are implied. However, it is crucial to ensure that someone following the algorithm does not wear multiple items of clothing if they are dumb.

A bug in your algorithm is an erroneous instruction(s) included in your steps that causes your algorithm to behave in a way you didn't anticipate.

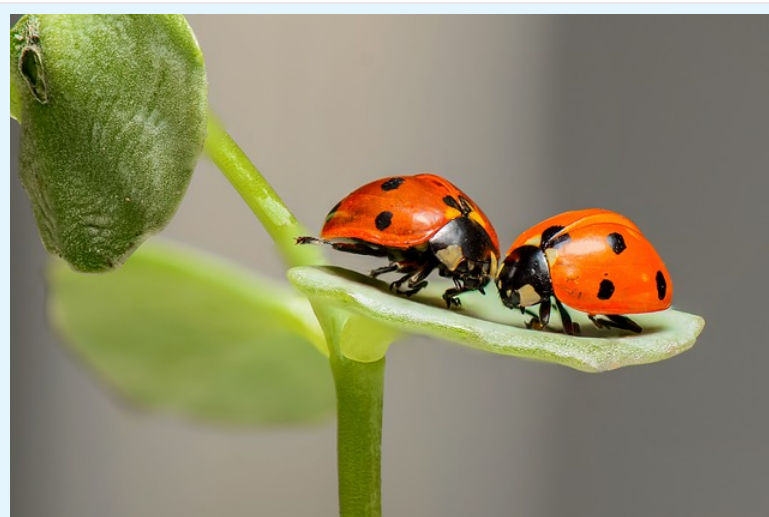


Figure 12.4: Always check for bugs in your code.

Remember, it might be useful to you to think about writing algorithms for a very, very, very dumb person. For example, consider the following Professor's Solution - Algorithm with the following, small, yet critical change (figure 12.5 below). Now, read it and pretend that you are exceptionally dumb and just follow the instructions blindly. Furthermore, let's say that the forecast is 61 degrees F. Before continuing reading the text, look at the picture below and see if you can find the bug.

```

1 Wake up from alarm at 6am
2 Take a shower
3 Check the weather to get the forecast
4 If forecasted temperature >= 70
5     Wear t-shirt and shorts
6 If forecast >= 50
7     Wear long-sleeve shirt and jeans
8 If forecast >= 30
9     Wear jacket and jeans
10 Else
11     Wear sweatshirt and slacks
12     Leave house at 7:30am, drive car to CSU
13 Leave house at 7:20am to ride bike to CSU
14 Lock bike safely
15 Arrive at class by 7:50am

```

Figure 12.5: Can you find the bug?

The dumb person MIGHT start at the top of the list, notice that the forecast is NOT greater than or equal to 70, deduce that the tabs imply that you should only follow the instructions if the preceding statement is true, then skip to line 6 (doubtful, but let's say that this particular dumb person is having a good day). When the dumb person gets to line 6, they notice that hey, `If forecast >= 50` is true so they continue and put on a long sleeve shirt and jeans. When they get to line 8, they notice that `If forecast >= 30` is ALSO true! So they put on a jacket and jeans over the long sleeve shirt and jeans they already have on. *That is the bug!* We didn't intend for someone to remember that an algorithm is a step-by-step process that should be written with nothing implied. Wear two pairs of jeans or two tops!

Decisions and Loops

Almost always, an algorithm needs to make a decision. We saw that in the 8 am attendance problem solution. The algorithm helped choose clothing based on the variable forecast. As mentioned earlier, computers are fast but really dumb. We can't just ask a computer what we should wear. We need to be specific in the steps it needs to follow to determine what we wear. Whenever we need to have a computer make a **decision**, we use an `if` statement. (e.g. `if forecast >= 60`)

Another thing that computers are really good at is doing boring, repetitive tasks over and over. For example, let's think about a computer program that assigns people to a military draft. Let's say that we need to draft 20,000 men into the armed services. We can write a computer program that randomly assigns all eligible men a draft number, then randomly chooses 20,000 numbers. If a human had to do this, it would take a very long time, but a computer can get this done very quickly! Whenever we need a computer to do a task over and over again (aka: **loop**) we use a `for` statement.

Decisions - `if` Statements

`if` statements are actually very simple and elegant ways to get computers to make decisions. They work by evaluating some conditional statements. If the statement evaluates to **true**, you do what follows. Optionally, you can include an **else** to specify what to do if the if statement evaluates to **false**.

Generically, if statements look like this:

```
If condition then do
sequence 1
Else do
sequence 2
```

For example, consider line 4 in the pseudocode in figure 12.5:

```
If forecast >= 70, continue, else go to line 7.
```

The decision is made based on the conditional statement. If the forecast temperature is greater than or equal to 70 degrees you continue, else you go to line 7. That is the decision! Hopefully, you know and understand how this works, and if you do, you are already thinking algorithmically!

Consider that a student designs an algorithm that informs them how much they should be studying each school day for an exam. The algorithm makes a decision about how much the student should study based on how far away the exam is. The student "runs" the algorithm once a day in the morning. Consider that the exam is 9 days away.

```
1 Check exam date
2 If exam is >= 3 weeks away, continue, else skip to line 5
3   Spend 15 minutes studying
4   Finish!
5 If exam is >= 2 weeks away, continue, else skip to line 8
6   Spend 30 min studying
7   Finish!
8 If exam is >= 1 week away, continue, else skip to line 11
9   Spend 1 hour studying
10  Finish!
11 If exam is >= 2 days away
12   Spend 2 hours studying
13   Finish!
```

Loops - for Statements

`for` statements allow us to write down repetitive actions algorithmically. Consider the generic `for` statement to get an idea of how it works:

`for` number of iterations

sequence repeats a number of iterations

The idea of a `for` statement is that you already have an idea of how many iterations need to be performed, then you algorithmically define the sequence that is to be performed beneath it. Let's return to the military drafting example to illustrate this. Consider the algorithm below that assigns all eligible men a draft number, then randomly chooses 20,000 numbers.

```
1 Assign all eligible men a draft number
2 number_eligible = total number of men eligible
3 For draft = 1 to 20,000
4     choose a random number between 1 and number_eligible
5     draft the person with the chosen number
```

Figure 12.6: *for* statement example algorithm

Looking at figure 12.6 above, hopefully, you can see why it is called a **loop**. Starting at the top, we first assign all eligible men a draft number. Let's say there are 150,000 eligible men, so the next line tells us to set the variable `number_eligible` to 150,000. On line three we have our `for` statement and *enter the loop*. The statement is telling us to start drafting at 1 and continue until 20,000. On line 4 we choose a random number, then on line 5, we draft the person with that number. Now the loop comes into play. Because we are in a `for` loop, we *automatically* increment `draft` by one and continue repeating lines 4 and 5 19,999 more times.

I understand that `for` loops can be tricky. Take some time to think about this section and if you are still confused supplement your reading with the [Wikipedia page on for loops](#). Don't skip your brain workouts!

Nested Statements

The last thing we should mention about algorithms is that you can *nest* statements within each other. For example, look at the following algorithm below in figure 12.7.

```
1 If country is at war, continue, else skip to line 8
2     Assign all eligible men a draft number
3     number_eligible = total number of men eligible
4     For draft = 1 to 20,000
5         choose a random number between 1 and number_eligible
6         draft the person with the chosen number
7     finish
8 Well there is no war going on so relax!
```

Figure 12.7: Example of nesting statements

Notice how we *nested* a `for` statement *within* an `if` statement. Nesting is a way to allow your algorithms to only loop if certain conditions are met (`for` within an `if`), or to make certain decisions a number of times (`if` within a `for`).

Again, this can be a confusing concept but once you start to use it in your programming it will make more sense. We will see the concept of nesting pop up in later chapters. For now, just know that it is an option.

Good Algorithm Checklist

All good algorithms have the following characteristics:

- Remember that an algorithm is a step-by-step process that should be written with *nothing* implied.
- Consider the variables that will be used in the algorithm. For example, in the 8 am the solution, the variable was forecast. Thinking about this ahead of time will make writing computer algorithms in MATLAB easier.
- Check for bugs. To do this it is best to pretend you are *really* dumb (this can actually be difficult to do). Make sure that the algorithm performs exactly the way that you want it to.
- If there are decisions or loops necessary, make sure that they are written out in such a way that they are easy to read and understand. Only use them if necessary.

Communicating Algorithms

There are probably other ways to communicate algorithms, but for this class, we will only consider two: pseudocode and flowcharts. I highly recommend getting used to both ways of communicating algorithms as they are often most useful when used in conjunction. I have worked at several engineering companies and I can safely affirm that engineers often use both pseudocode and flowcharts in conjunction.

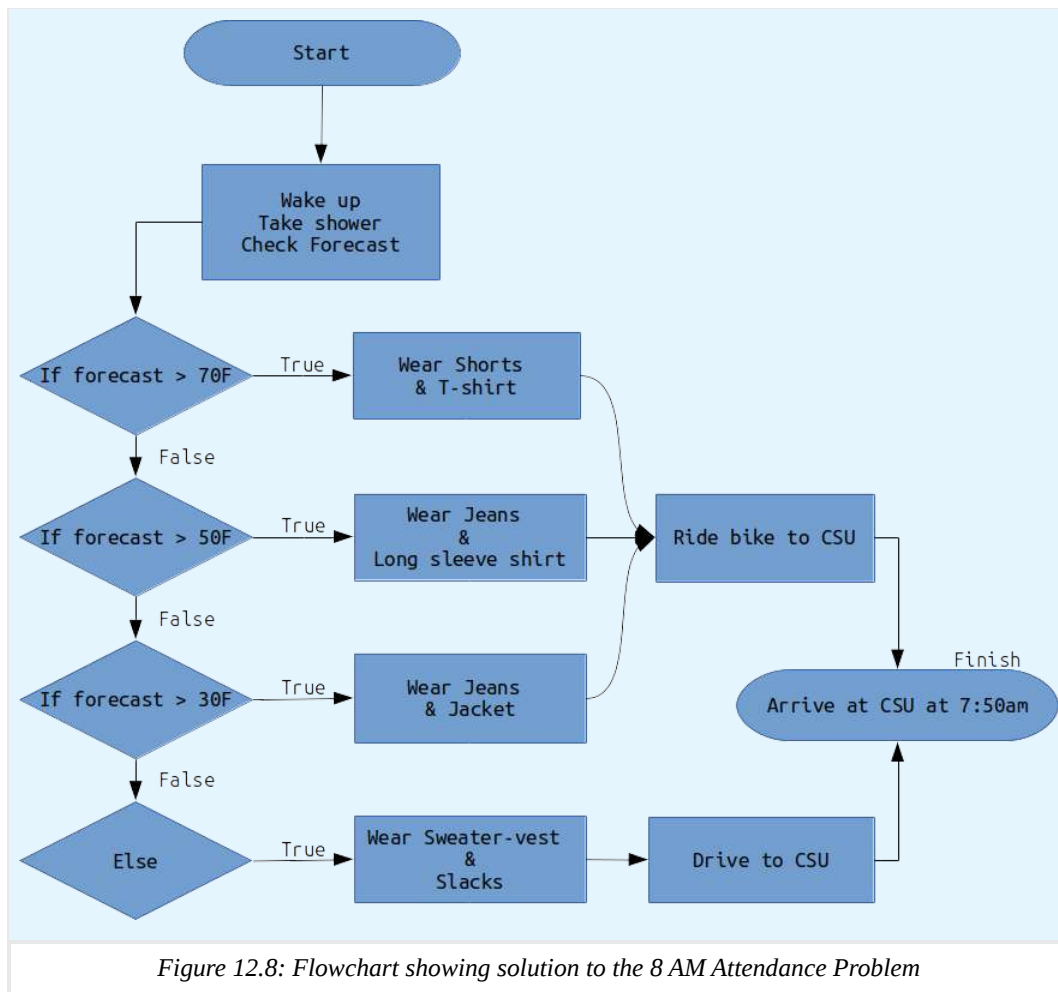
The last thing to note is that there are no *rules* when it comes to pseudocode or flowcharting there are just guidelines. I once had a student that used to draw elaborate pictures for her “flowcharts”. Elephants meant something I can’t remember, but it was crazy. Her technique worked for her, but it was impossible for anyone else to follow. The guidelines that I will present below are standardized and should be understandable by your professors, tutors, or TAs.

Pseudocode

Lucky for us we have already seen examples of pseudocode. **Pseudocode (seen in figures 12.3, 12.5, 12.6, 12.7) is just an implementation of an algorithm in plain English.** Writing pseudocode can be helpful because you *think* about how your algorithm will work before you start to worry about programming it. The only real rule for pseudocode is that it needs to describe the logic of the algorithm so that when you go to translate the algorithm into a computer programming language, it is simply a task of changing English to MATLAB.

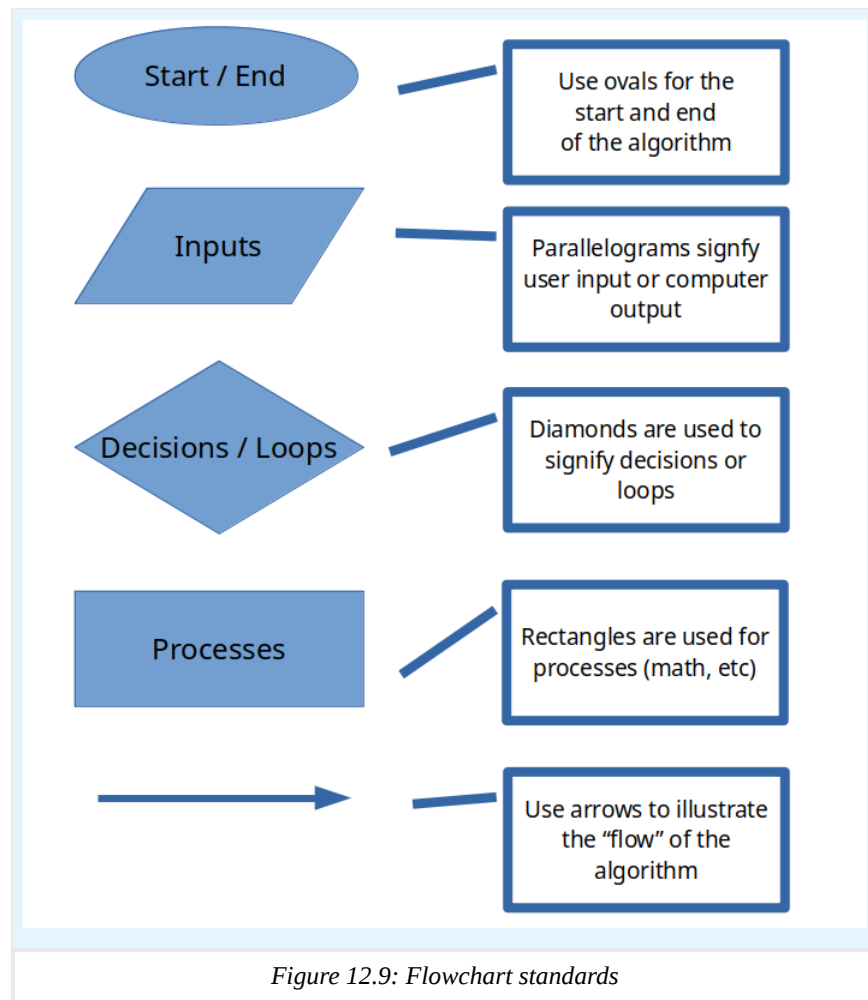
Flowcharting

Another common way to write down algorithms is using a flowchart. Before we talk about some general rules for flowcharting, look at the flowchart for the 8 am problem shown below in figure X. I think the best way to learn about how to flowchart, is to look at a flowchart and discover its logic on your own.



As you can see, flowcharts are nice because they are pictures with arrows which makes it easy to follow along. Just like your pseudocode, in order for flowcharts to be effective, they need to fully describe the logic of the algorithm. Again, that makes copying the algorithm into MATLAB (or whatever programming language) much easier.

There are some generally accepted standards for flowcharting presented below in figure 12.9.



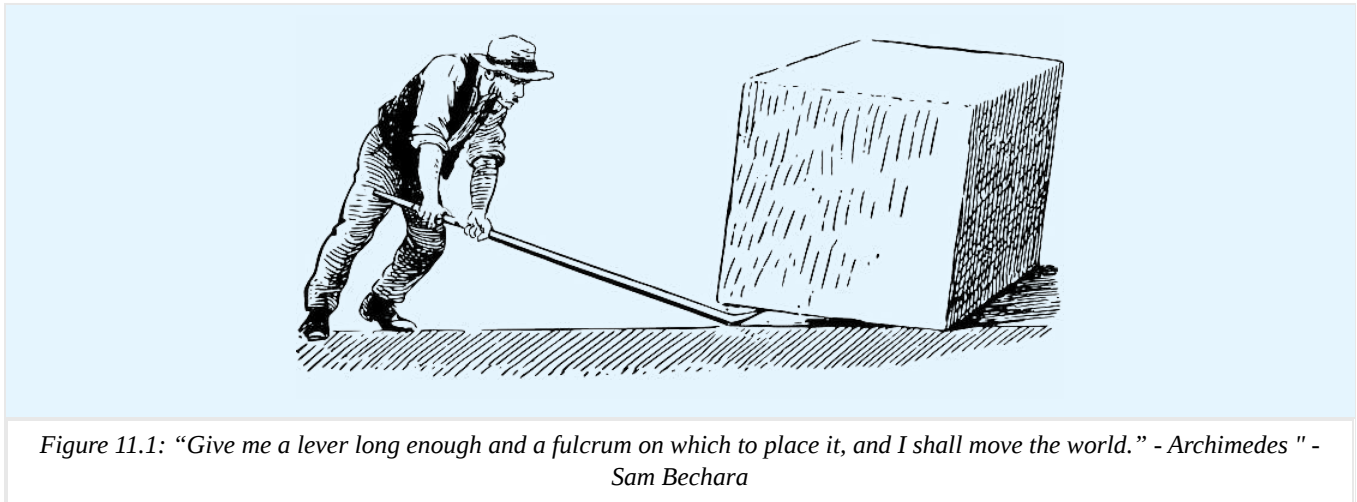
The only real rule with flowcharts is that they are useful to you. Honestly, I prefer flowcharting to pseudocode but in my professional life, I use both to help organize my thoughts and force me to think algorithmically. Whichever you decide to use is a matter of personal taste, but all good engineers *will use at least one of these techniques*.

Think about an algorithm that only you would know (or very few of your classmates), and explain it “algorithmically” as either a flowchart OR pseudo-code. The only requirement is that it should have some decisions to make and should contain no bugs. My suggestion is to avoid recipes, they are kind of boring and aren’t really unique TO YOU. Perhaps you are a gifted gardener and know exactly how to prune apple trees? Are you into repairing cars and know the best procedure to follow to diagnose and replace an alternator? I am willing to bet that there is something that you know how to do that a lot of your classmates would not understand (it doesn’t matter how nerdy or weird it is).

This page titled [3.1: Thinking Algorithmically](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 12: Thinking Algorithmically](#) is licensed [CC BY-NC 4.0](#).

3.2: Relational and Logical Operators



This chapter is a little weird. We are going to learn about relational and logical operators but it will seem out of place because we will not learn how to apply them until the next chapter. Eventually, we will use what we learn in this chapter to give our programs the ability to *flow* and *make decisions*. Until then, concentrate on learning how these operators work with my promise that it will make more sense why they are important in the coming chapters.

Before we jump in, a quick warning on this chapter. It is *really easy* to skip your brain workouts for this chapter. It is so tempting to type everything into MATLAB to get the answers without thinking. I urge you not to do this! When evaluating relational and logical expressions it is important to understand how they work in order to use them correctly. Be honest, follow along with this text, and be sure to not skip your brain workouts! Without further ado...

Learning Goals

In this chapter, we will learn what the relational and logical operators are and how they work. We will save for *why* we need them until the next chapter. For now, we will learn:

- What the relational and logical operators are and how do they work.
- What the order of precedence is and why it matters.
- How to evaluate statements with mathematical, relational, and logical operators.

Relational Operators

Before we can start creating programs that can make decisions we have to learn the framework to have our programs start comparing entities. That is where the relational operators come in. The relational operators look the same as math inequalities and they work in a similar fashion. The difference is, with math you are typically looking to “*solve*” the inequality, when we use the relational operators, we are simply interested if the operation is **True** or **False**.

For example, consider the following:

$$7 + 2 > 8$$

Does it evaluate to **True** or **False**? Well since $7 + 2 = 9$ and $9 > 8$ this particular statement evaluated to **True**! We will learn how this helps our programs make decisions in the sections below. For now, the important thing is to understand what the relational operators *are* (table 11.1 below) and how they are used in MATLAB.

Symbol	Role
<code>==</code>	Equals
<code>~=</code>	Not Equal
<code>></code>	Greater Than
<code>>=</code>	Greater Than or Equal
<code><</code>	Less Than
<code><=</code>	Less Than or Equal

Table 11.1: The Relational Operators

The relational operators can be used to compare variables, combinations of variables and numbers, or even mathematical operations. The important thing to remember is that when the operation is completed, either **True** or **False** comes out.

The only ones that might not seem intuitive are `==` and `~=` but again, they are very similar. For example, consider the following:

```
>> 4 == 4
```

Since 4 does in fact equal 4, the preceding statement equals **True**. Nothing groundbreaking here. Let's consider one more:

```
>> 4 ~= 4
```

In table X above you can see that the `~=` operator is the not equal operator. In this case, 4 does equal 4 so the statement 4 is not equal to 4 is **False**.

True and False

We saw above that the relational operations evaluate to either **True** or **False**. In computer science **True** and **False** are referred to as **booleans**. That isn't super important to remember for now. Instead, the important thing to learn is that, unlike other programming languages, **MATLAB is extremely flexible and allows both True and False to be represented in a number of ways.**

In MATLAB, the keyword `true` can be used or the number `1` can be used to represent **True**. Similarly, the keyword `false` and the number `0` can be used to represent **False**.

Try It!

The best way to understand what I am trying to say is to try it yourself. So fire up MATLAB and let's investigate how the relational operators work. The brain workout part of this is that *you should stop and think about what should happen before you run any of the following commands!* It will be impossible for anyone to know if you skip this step or not but do not skimp on your brain workouts!

To begin, let's define a few variables in the workspace to work with (we will use them in other examples so it is a good idea to keep them around):

```
1 tomato = 23;  
2  
3 cucumber = -12;  
4  
5 salmon = 7 * 8;
```

runrun allrestartrestart & run all

Next, we can use those variables to investigate how the relational operators work. For example, try executing the following operation in the command window. Before you hit enter, what do you **think** should happen?

```
1 tomato = 23;  
2  
3 cucumber = -12;  
4  
5 salmon = 7 * 8;  
6  
7 tomato > cucumber
```

run run all restart restart & run all

Hopefully, you aren't jumping ahead and are actually taking a moment to think. In any case, you should notice that MATLAB assigns the automatic variable `ans` the value of `1` which is to say that the operation is evaluated as `true`. That makes sense since `tomato` is a variable storing the number `23` and `cucumber` is storing `-12`, it is clear that $23 > -12$ and we would expect MATLAB to think this is **true**.

We can also *assign the output of a relational operation to a variable*. To see this in action try:

```
1 tomato = 23;  
2  
3 cucumber = -12;  
4  
5 salmon = 7 * 8;  
6  
7 result_1 = salmon < tomato
```

run run all restart restart & run all

```
>> result_1 = salmon < tomato
```

Notice that MATLAB creates a new variable in the workspace called `result_1` and that it is storing the number `0` which is the same thing as `false`. That is what we would have expected the following operation to evaluate to.

The Order of Precedence



Figure 11.2: MATLAB operators have to wait for their turn.

There is actually one more important way that we can use relational operators, we can use them in combination with mathematical operators. For example, we COULD perform an operation like this.

```

1 tomato = 23;
2
3 cucumber = -12;
4
5 salmon = 7 * 8;
6
7 pie = pi * 2;
8
9 pie + salmon^2 > tomato * salmon + 2

```

run

run all

restart

restart & run all

The problem is that we have not discussed [the order of precedence](#) (link is to MATLAB's description). This is very similar to the mathematical concept of the order of operations. The only difference is that the order of operations does not include the relational or logical operators (we will talk about them later). Just like the order of operations, the order of precedence lets us know what mathematical, relational, and logical operation steps will happen in what order. MATLAB's order of precedence is presented in table 11.2 below.

You can think about it like the operators are waiting in line to get into a club. The bouncer is going to let everyone in but they don't all get to come in first come first serve. There is a hierarchy, and that hierarchy is listed in table 11.2 below.

Order Number	Operator	Name
1	()	Parenthesis
2	.^	Power
3	~	Logical Negation
4	.* ./	Multiplication, Division
5	+ -	Addition, Subtraction
6	:	Colon Operator
7	< <= > >= == ~=	The Relational Operators
8	&	Logical And
9		Logical Or

Table 11.2: The MATLAB Order of Precedence

You should note that this is slightly different than what MATLAB presents as the order of precedence, I truncated the [official list](#) down to what we will be using in this book. Let's take another look at that MATLAB statement we started this section with:

```
>> pie + salmon^2 > tomato * salmon + 2
```

Now we can see how to apply the order of precedence and evaluate this statement! **The best way to solve these types of statements is to reduce according to the order of precedence and then continue.** For example:

1. There are no parenthesis so we can skip that
2. We need to evaluate the power. Since we have salmon^2 we can evaluate that first. $\text{salmon} = 7*8$ implies that $\text{salmon}^2 = 56^2 = 3136$
3. Now our statement has reduced to: $\text{pie} + 3136 > \text{tomato} * \text{salmon} + 2$ Next on the order of precedence is $. * .$. We have one multiplication operation, $\text{tomato} * \text{salmon} = 23 * 56 = 1288$.
4. Now our statement has reduced to: $\text{pie} + 3136 > 1288 + 2$ We now perform the math operations. Recall $\text{pie} = \text{pi} * 2$ which is ≈ 6.2832 . So... $\text{pie} + 3136 = 6.2832 + 3136 = 3142.2832$ and $1288 + 2 = 1290$
5. We are almost done. At this point we have: $3142.2832 > 1290$ Lastly, we have a relational operator, $>$. We are now ready to evaluate it!

Hopefully, you nailed Question 11.2 above. Let's try one more little brain workout before we move on and discuss the Logical Operators.

The Logical Operators

The last thing we need to learn before we can start having our programs make decisions is how to combine multiple relational operators together. Although it is jumping a little ahead, let's consider *why* we need the logical operators.

Consider designing a piece of software that needs to make a decision about a range. For example, I personally ride my bike to work but won't ride if the forecasted high is above 35 °C or if the forecasted low is below -10 °C. If I wanted to make an app that could tell me if I should ride my bike to work or not, I would need a program that could determine if: the forecast high is less than 35 °C `AND` forecast low is greater than -10 °C. The `AND` in the previous statement is formatted in code because it represents where we would have included a `logical and`.



Figure 11.3: You know who else is into logic? These old boys.

The logical operators give us a way to string together multiple relational statements and get one **true** or **false** out. When looking at the order of precedence you may have noticed the three logical operators, shown in table X below. **Take a minute to look at table X below and read the description of each of the logical operators.**

Symbol	Name	Description
~	Logical Not	>> ~A Not only operates on one operand. It gives the opposite Boolean value. So if A is true, ~A is false .
&	Logical And	>> A & B If A and B are both true , evaluates to true . Otherwise, it evaluates to false .
	Logical Or	>> A B If either A or B are true , the statement evaluates to true . It evaluates to false only in the case where both A and B are false .

Table 11.3: The logical operators

You can see that the logical operators `&` and `|` allow us to string together multiple relational operations! Cool! Consider the following MATLAB command:

```
1 6 >= 7 & 5 >= 6
```

run
run all
restart
restart & run all

According to the order or precedence, we **evaluate the relational operators first**. That means that the statement reduces to:

```
>> false & true
```

Looking up in table X for the correct way to use the logical operator, we can see that `false & true` evaluates to **false**!

Logical `&` is the type A logical operator. It needs *all* the relational operators to evaluate to true in order to return true. If logical `&` was a person they would be kind of uptight but would probably be very successful. Only the truest of the true for logical `&` !

Logical `|` is the cool guy logical operator. If *any* of the relational operators evaluate to true, it calls the whole thing true. Logical `|` probably drives a Volkswagen bus and says “rad” a lot. All logical `|` cares about is at least one truth.

In the future, we will see how both `&` and `|` are critical to creating complex programs. Just like yin and yang, we need both `&` and `|` to make our programs work. Logical `&` and logical `|` are a little tricky but once you understand the logic, you will be able to make some cool programs.

A Short Note on Logical Not ~

Logical `~` is a little weird because I find it redundant and unnecessary. Why? If you have a statement like:

```
1 ~(4>5)
```

runrun allrestartrestart & run all

We can see that the `(4>5)` will evaluate as false. Then `~false` is **true**. Ok, but why not just put `4<5` ? See what I am getting at? If you just reverse your relational statement, you shouldn't need to flip your Boolean value. There might be a good reason that I have not yet run into for using a logical not. If you run into a good and necessary use of logical not, please email me and let me know (samuel.bechara@colostate.edu)!

Putting It All Together

Using the order of precedence, you should now be able to evaluate statements with mathematical, relational, and logical operators all in tandem. The trick is to reduce the statement using the order of precedence until you are left with a single Boolean value.



Figure 11.4: A lovely tandem

Let's consider one more example together to tie it all together:

```
1 the_answer = 3 == 3 & (~ (42 == 21 * 2 | 65-2 == 65 + 3))
```

run

run all

restart

restart & run all

Try It!

Before you move on to see my solution, make sure you try to solve it on your own! Remember, use the order of precedence, and reduce the statement until you are left with one Boolean value. You can do it! It is important to not skip your brain workouts!

Ok, now that you have your solution, I will show you how **I** would solve this problem:

Since parenthesis are first on the order of precedence, we will start with the innermost set and reduce this chunk first:

```
( 42 == 21 * 2 | 65-2 == 65 + 3 )
```

According to the order of precedence, we will want to perform the multiplication step first. I am also going to show you a shortcut.

```
( 42 == 42 | <some other relational operation> )
```

Why did I do the `21*2` but skip everything on the right side of the logical `|` ? Remember, logical `|` is chill, if either of the statements being compared evaluates to **true**, the whole thing is **true**. Since `42 == 42` is **true**, it doesn't matter what `<some other relational operation>` evaluates to! Returning to our original statement we have:

```
>> the_answer = 3 == 3 & (~ true)
```

The next step would be to evaluate the parenthesis. In this case `~true` is **false** so we reduce it to:

```
>> the_answer = 3 == 3 & false
```

We are almost done! Here we could take another shortcut even though the `3 == 3` is easy. Remember, logical `&` is uptight. Since we have a **false** being compared with a logical `&`, it doesn't matter what `3 == 3` evaluates to (hopefully it is obvious to you that `3 == 3` is **true**)! The whole thing is going to turn out false!

```
>> the_answer = false
```

Disclaimer

The reality is that you will never have to solve problems like this in the real world. Instead, you will be asked to utilize relational and logical operators in tandem to create complicated programs that can react intelligently to different inputs. That is what our programs have lacked so far! The ability to make decisions. Hopefully, now you can see how combinations of these operators might allow us to control the flow of our programs in the future.

This page titled [3.2: Relational and Logical Operators](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 11: Relational and Logical Operators](#) is licensed [CC BY-NC 4.0](#).

SECTION OVERVIEW

3.3: Documentation and such

Documentation

Why have a chapter on documentation? Because documentation is needed for any program you write, however, it is also the most avoided essential. *Why?* Because of the oft used excuse that "I know what it does." But unfortunately you forget after awhile...so...**DOCUMENT!!!!**

Documentation is one of the most important parts of programming, it is also the toughest, and most poorly implemented in general. Here we will present some general rules for documenting any program (software project).

- Documentation for all programming projects
 - A traditional documents on paper (or pdf or html or some universal way to read the document)
 - Manual
 - One page quick guide
 - Paper (or papers) explaining the raison d'etre for your program and the theory and solutions
 - Documentation within the program itself
- **All** programming languages have some method of allowing the user to enter comments
 - Comments should be added at the beginning of the program
 - Comments should be throughout the program itself near *important* lines
 - Don't overdo comments as this distracts from the overall documentation
 - Example: we don't need a comment on a print statement that says it displays the string to the screen...we can figure this out without documentation
 - Example: you use a special numerical method that is not very common...yes please document this
 - Octave and its ilk note: the **help** in Octave give the front comments; **lookfor** give the first line
 - So if you don't have the front comments, help won't work...the professor will not be pleased...
 - Front comments can also be where you credit yourself (and your partners if applicable)
- Good coding techniques acts as a defacto documentation of the code
 - Use tabs for statements in loops and conditionals, etc.
 - Note you have to do this in Python (so you are force to have good coding techniques)
 - A lot of editors (for coding) do this for you as long as you have a template for that language

- Separate functions that are used multiple times into function, subroutines, or the like

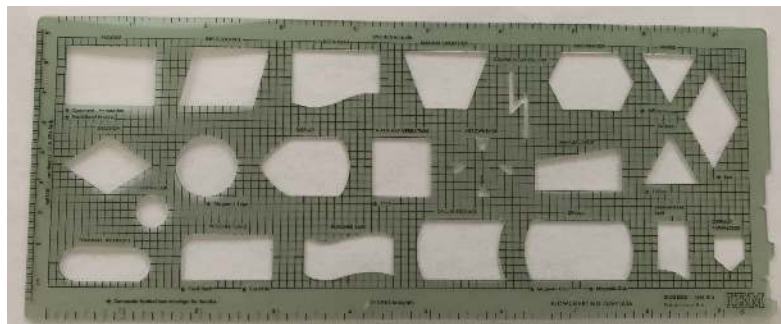
[illegible]

- Documenting in a group project
 - Documenting in a group project requires some negotiation and sharing of responsibility
 - It is required of engineers to work in groups
 - It is required of instructors to grade you on group dynamics
 - Extreme Documentation
 - Characteristics
 - Document as project progresses (not after it is done)
 - Team ownership of project
 - Independent documentation from each programmer in the group (more eyes theory)
 - Compile documentation at the end in a traditional document and in the program itself

Writing Good Programs

To write good programs you should follow the procedures on documentation above, but *what about the actual code?* It is generally agreed that you should outline the procedure before you begin to actually code your program. There are two methods for outlining a program, flowcharts which has actually done before computer programming but adopt by programmers as well, and pseudocode where you code in a generic sense and to the specific syntax after you have gone over the pseudocode.

- Flowcharts
 - Set of graphic "blocks" with connection arrows that represents the programming flow in a visual way
 - There has been work on using set theory to actually prove if a program is valid using flowcharts
 - Just proves you get a result and it follows you procedure as you believe would work
 - Can't really prove if the result is right or not as that takes an expert opinion in the field that the programming is being written for
 - There are programs that help with flowcharting or even run programs that you have flowcharted (*raptor* or *flowgorithm*)
 - There are modern additions that allow for parallel processing in flowcharting
 - Flowcharts have official ANSI symbols
 - While an old technique it is still useful



Flowchart symbols on a template that was used before personal computers.

- Pseudocode
 - Programming in plain English (or any other language you wish) with a limited set of rules
 - Takes away the burden of strange (in the sense of our normal English sensibility) variable names, structures, methods, etc. imposed by different computer languages
 - Lets us write the way we "think"
 - Some pseudocodes have been developed with hardware and software interaction in mind
 - Software may actually be involved in changing voltage levels (think a controller) which is hardware in nature but controlled by software
 - Pseudocode is a generally enough idea that it can be used things not necessary computer programming related - just like flowcharts

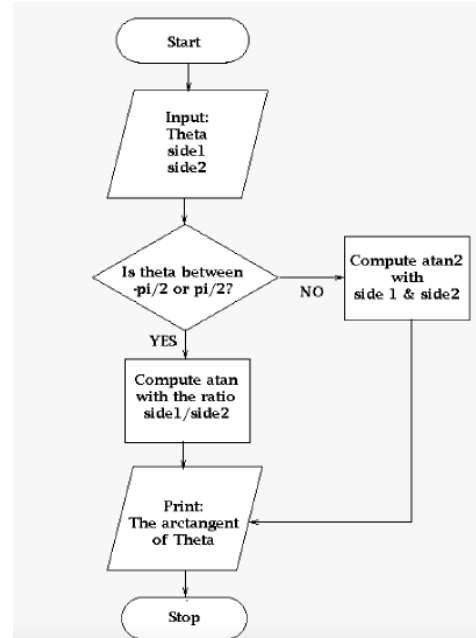
- While generally pseudocode is what you like to say in English it is generally agreed on that it should have words similar to programming languages
 - Might use read and print in pseudocode for a C++ program and those "pseudocode" words would be changed to cin and cout when you start to program
 - There is no official pseudocode

Comparison of pseudocode and flowchart with very simple routine

```

input theta,side1,side2
if (-pi.2 < theta < pi/2) then
  ... answer = atan(side1/side2)
else
  ... answer = atan2(side1,side2)
endif
print answer
return
end
  
```

Pseudocode of a simple routine to decide whether to use atan or atan2. Note there is input and output and a conditional.



Flowchart of a simple routine to decide whether to use atan or atan2. Note there is input and output and a conditional.

3.4: Programming in any language

Programming

To review, the purpose of programming on an electronic computer is to solve problems in a timely manner that are a difficult to solve by hand. Here we will review five languages as our examples: Octave (MatLab/Scilab), IDL¹ (PV-Wave or GDL), Python, Fortran, and C++, with a heavy emphasis on Octave (the other languages are included to show similarities between these languages, as most languages are very similar).

While you could, as examples, use Octave, Python, or IDL to just add, subtract, divide, or multiply numbers as a sort of calculator that is not the best use of programming languages. A calculator would be better for that sort of work. What makes programming languages powerful is variables, not dissimilar to variables in regular mathematics, especially "variables" that are structures which are reminiscence of matrices or matrices themselves (see parachute person).

We will use Octave for the examples herein and other chapters to show what is being discussed in the text. Please run them and feel free to modify them and run them again (NB: your modifying the programs does not effect them for others...just you. So feel free to play.)

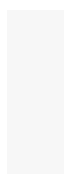
- Here we do simple math in Octave which makes Octave a great calculator, but does not nearly demonstrate its true power.

```
1 2+2
2 2*2
3 (2+2i)*(2+2i)
4 sind(180)
```

runrun allrestartrestart & run all

- Here we do simple math in Octave but add a variable component. This increase the power but still doesn't get to the full power of a programming language.

```
1 x = 0;
2 x = input("Enter degrees in radians ");
3 printf("The sine of %f is %f\n",x,sin(x))
```



run run all restart restart & run all

In this last example we use a variable which we will discuss in the next mini-section.

Numbers, variables, arrays, and other structures

In general we would like our computer programs to provide the user with a method to do either **complicated** calculation, complicated string matching and pattern matching, or a combination of both of these. Everything a computer does we can do by hand but with less ease (though some might argue after struggling with a program that that is not true, but as your skill grows you will find that indeed it is true). In order to do these complicated calculations there must be a way to have and use variables. Variables in a computer languages work just like they do in mathematics (and other fields)...a way of expressing an unknown or many unknowns.

Variables for calculations in computer languages are either declared as or given a specific type of number. A number (a scalar) in mathematics can be a counting number, an integer, a real number, a complex number, etc. In computers we have similar designations though the syntax differs depending on the programming language. The very basic **data types** are shown for a set of chosen languages below.

Basic data types	Fortran (mostly required)	C++ (required)	Python (doesn't exist)	Octave/Matlab (doesn't exist)	IDL (optional)	Your math class
Integers	integer	int	Python does this for you	Octave does this for you	int	integers
Real	real	float, double	Python does this for you	Octave makes all numbers <double>	float, double	real numbers
Complex	complex	<none>	Python does this for you (using j as the imaginary number)	Octave does this for you (using either i or j as imaginary number)	<none> (Need to use a function)	complex number (real and imaginary)
String	character (len=#)	char var_name(len)	Python does this for you	Octave does this for you	string	words

There are many other data types that you will learn as you become a more sophisticated programmer, but for now these are a good start. It is likely that for engineering and science programs a programmer will need to use matrices (or structures) with the numbers having a specific type like in the above table. For equations most programming languages allow you to write it just as you would in mathematics, however just solving one equation still is more like a calculator rather than a programming language. For more utility a programming language would allow you to loop, do conditional branches (if A do B or else do C), and input and output data. These type of statements are the program control statements. But first let us introduce operators and functions (which we have already used above but let's complete the discussion).

Computer Operators

Programming in any language requires operators (like + and -) to do useful functions. has three main control statements.

- Just like in mathematics there are operands and operators
- For most languages computer operators follow the same precedence as mathematics operators

- Below we will briefly look at three of the most common operators

→ **Arithmetic operators**

→ **Relational operators**

→ **Logic operators**

Computer Functions

Programming in any language has functions like print or write.

→ **Functions**

→ **Subroutines**

→ **Script**

Control Statements

Programming in any language has three main control statements.

→ **Loops**

→ **Conditionals**

→ **Input and output**

Below is a script example of each of the major control statements²

- This is an example of a loop.

```
1 for i = 1: 10
2     printf('This is a loop in Octave (or Scilab or Matlab)...counting numbers %i\n',i)
3 endfor
```

run

run all

restart

restart & run all

- This is the example of a conditional statement, however it does have input and output as well.

```
1 yn = input("Is engineering fun? (Y/N)", "%s");
2 if (yn == "Y" || yn == "y")
3     printf("Yes engineering is fun!!!");
4 else
5     printf("Hmmm...");
6 endif
```

run run all restart restart & run all

- This is the example for input and output, however it does have a conditional statement as well.

```
1 snumber = input("The input statement allows us to input numbers or letters. How many suns do we have?")
2 if (snumber == 1)
3     printf("That is correct.\n");
4 else
5     printf("An astronomy course is recommended for you.\n");
6 endif
7 name = input("My name is ", "%s");
8 printf("Hello %s, as you can see the printf statement is a form of output.\n", name);
```

run run all restart restart & run all

So we have used Octave (as our example program) as a fancy calculator (which has its uses) but we have not used the true power of Octave. The true power of a programming language is to write a program (or function).

- This is an example of a function that has input and output, a conditional, and some loops. This is a different example of input and output where it uses a file (you are more likely to use files in "real world" work). Normally this program is saved in a file with an extension of *name_of_program.m* (or in this case *readwritefile.m*) and when you run it in octave you just run `readwritefile()`, but in this Libretext format we have the file a special area so you can see it, edit it (yes you can edit this - it won't be permanent) and run it.

```
1 %
2 % Reading and writing files ("advanced" input and output)
3 %
4 function nul = readwritefile()
5 %
6 %
7 num1 = [1,2,3];
8 num2 = [];
9 sznum1 = size(num1);
10 %
11 % Note there are many commands to write a file in octave including
12 % fprintf (which we use here), fputs, fwrite, dlmwrite, etc...
13 % this does not include octaves own method of
14 % saving matrices (for a much faster access) of just "save"...the better way to go for
15 % mathematical work...
16 %
```

```

17 fidw = fopen("testfile","w");
18 for i = 1:sznum1(2)
19     fprintf(fidw,"%d\n",num1(i))
20     disp("Wrote in testfile"), disp(num1(i))
21 end %for
22 fclose(fidw);
23 disp("File written and closed")

```

run

run all

restart

restart & run all

There are multiple languages to write programs in but herein we will just detail five languages. In order to do this we will show examples of the same routine written in different languages, starting with Octave (because it can run on in this format and it is an excellent language for engineers).

Example of different languages

In this section we will compare five languages (Octave, GDL, Python, Fortran, and C++) using the same routine for calculating a root of a number $ans = \sqrt[n]{A}$. First the function as a simple Octave routine (this is not for comparison) as it should be written, but is not as good for an example.

```

1 % Function to figure out roots: A^(1/n)  x0 is initial guess;
2 % tol is the tolerance that we are willing to accept
3 % Example: octave:2> root = nroot(20,3,2.2,1e-6)
4 %           root = 2.7144
5 %
6 function root=nroot(A,n,x0,tol)
7 x1=x0;
8 x2 = 1/n*[(n-1)*x0+A/(x0)^(n-1)];
9 while (abs(x2-x1) >= tol)
10     x1=x2;
11     x2 = 1/n*[(n-1)*x1+A/(x1)^(n-1)];
12 end %while
13 root=x2;
14 return % root
15 end
16
17 root=nroot(20,3,2.2,1e-6)

```

run

run all

restart

restart & run all

Here we have a function that has all the inputs in the function itself, rather than asking the user to enter them one by one. In the next example we will ask questions rather than putting the inputs in the function. Doing this makes the program more interactive but it does not make it very reusable, so this is mostly for an academic exercise. If you were to use a function use the one above.

```

1 % Function to figure out roots: A^(1/n)
2 % Need a tolerance for how close you want the answer to be
3 % And an initial guess
4 %
5 % This is a demo program to show control structures in action while solving
6 % a math problem (the nth root of A). While most programming languages
7 % can do that without a program being written this is a method to exemplify
8 % writing simple program.
9 %
10 function nul=control_root()
11 base = input("What is your base? ");
12 degree = input("What is the degree for the root you want? ");
13 tol = input("What tolerance do you wish to work towards? ");
14 guess = input("You will need an initial guess, what would it be? ");
15 answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));
16 while (abs(answr-guess) >= tol)
17     guess = answr;
18     answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));

```

```
19 end %while
20 printf("The answer is %g\n",answr);
21 return
22 end
23
24 control_root()
```

run

run all

restart

restart & run all

Note that the comments section (indicated by the %) are useful in Octave as that is where the help comes from.

```
octave:1> help control_root
'control_root' is a function from the file /howtos/exampleprograms/
control_root.m

Function to figure out roots: A^(1/n)
Need a tolerance for how close you want the answer to be
And an initial guess

This is a demo program to show control structures in action while solving
a math problem (the nth root of A). While most programming languages
can do that without a program being written this is a method to exemplify
writing simple program.
```

To compare the programming languages to each other we will first present the program and how it runs, then compare the programs to either Octave or Fortran.

Octave

```

1 % Function to figure out roots: A^(1/n)
2 % Need a tolerance for how close you want the answer to be
3 % And an initial guess
4 %
5 % This is a demo program to show control structures in action while solving
6 % a math problem (the nth root of A). While most programming languages
7 % can do that without a program being written this is a method to exemplify
8 % writing simple program.
9 %
10 function n1=control_root()
11 base = input("What is your base? ");
12 degree = input("What is the degree for the root you want? ");
13 tol = input("What tolerance do you wish to work towards? ");
14 guess = input("You will need an initial guess, what would it be? ");
15 answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));
16 while (abs(answr-guess) >= tol)
17     guess = answr;
18     answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));
19 end %while
20 printf("The answer is %g\n",answr);
21 return
22 end
23 --
24

```

The program above is written in Octave, we have input and output, and a conditional and loop all in one (*while* statement).

```

GNU Octave, version 6.2.0
Copyright (C) 2021 The Octave Project Developers.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.

Octave was configured for "x86_64-apple-darwin20.3.0".

Additional information about Octave is available at https://www.octave.org.

Please contribute if you find this software useful.
For more information, visit https://www.octave.org/get-involved.html

Read https://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1> control_root()
(What is your base? 20
What is the degree for the root you want? 3
What tolerance do you wish to work towards? 1e-6
You will need an initial guess, what would it be? 2.2
The answer is 2.71442
octave:2> root=nroot(20,3,2,2,1e-6)
root = 2.7144
octave:3>

```

This is the run of the program on the left. First Octave itself has to be executed and then the program called `control_root` is executed in the Octave program.

```

1 % Function to figure out roots: A^(1/n)
2 % Need a tolerance for how close you want the answer to be
3 % And an initial guess
4 %
5 % This is a demo program to show control structures in action while solving
6 % a math problem (the nth root of A). While most programming languages
7 % can do that without a program being written this is a method to exemplify
8 % writing simple program.
9 %
10 function n1=control_root()
11 base = input("What is your base? ");
12 degree = input("What is the degree for the root you want? ");
13 tol = input("What tolerance do you wish to work towards? ");
14 guess = input("You will need an initial guess, what would it be? ");
15 answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));
16 while (abs(answr-guess) >= tol)
17     guess = answr;
18     answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));
19 end %while
20 printf("The answer is %g\n",answr);
21 return
22 end
23 --
24

```

Comparing the Octave program here to the Fortran program on the right shows some differences but a number of similarities. Octave does not have data declarations and combines the *print* and *read* statements (into *input*). But the equation is the same (except for the power symbol) and there is a similar *while* statement.

```

1 PROGRAM CONTROL_ROOT
2 C
3 C Function to figure out roots: A^(1/n)
4 C Need a tolerance for how close you want the answer to be
5 C And an initial guess
6 C
7 C This is a demo program to show control structures in action while solving
8 C a math problem (the nth root of A). While most programming languages
9 C can do that without a program being written this is a method to exemplify
10 C writing simple program.
11 C This is written in Fortran --- use gfortran if you have install it
12 C
13 C
14 C
15 C double precision :: degree, base
16 C double precision :: tol
17 C double precision :: guess
18 C double precision :: answr
19 C print *, 'What is your base? '
20 C read *, base
21 C print *, 'What is the degree for the root you want? '
22 C read *, degree
23 C print *, 'What tolerance do you wish to work towards? '
24 C read *, tol
25 C print *, 'You will need an initial guess, what would it be? '
26 C read *, guess
27 C answr = 1/degree*((degree-1)*guess+base/guess**(degree-1))
28 C do while (abs(answr-guess) >= tol)
29 C     guess = answr;
30 C     answr = 1/degree*((degree-1)*guess+base/guess**(degree-1))
31 C enddo
32 C print *, 'The answer is '
33 C print *, answr
34 C stop
35 C end

```

Comparing Fortran to the Octave program on the left has some differences, but is essentially the same. The data declarations being the biggest difference.

GDAL (IDL)

```

1 ; Function to figure out roots: A^(1/n)
2 ; Need a tolerance for how close you want the answer to be
3 ; And an initial guess
4 ;
5 ; This is a demo program to show control structures in action while solving
6 ; a math problem (the nth root of A). While most programming languages
7 ; can do that without a program being written this is a method to exemplify
8 ; writing simple program.
9 ;
10 pro control_root
11 print,"What is your base? "
12 read,base
13 print,"What is the degree for the root you want? "
14 read,degree
15 print,"What tolerance do you wish to work towards? "
16 read,tol
17 print,"You will need an initial guess, what would it be? "
18 read,guess
19 answer = 1/degree*((degree-1)*guess+base/guess^(degree-1))
20 while (abs(answer-guess) GE tol) do begin
21     guess = answer
22     answer = 1/degree*((degree-1)*guess+base/guess^(degree-1))
23 endwhile
24 print,"The answer is ",answer
25 return
26 end

```

The program above is written in GDL, we have input and output, and a conditional and loop all in one (*while* statement)

```

GDL - GNU Data Language, Version 0.9.9
- For basic information type HELP./INFO
- Default library routine search path used (GDL_PATH/IDL_PATH env. var. not set)
:
: /opt/local/share/gnudatalanguage/lib
- No startup file read (GDL_STARTUP/IDL_STARTUP env. var. not set).
- Please report bugs, feature or help requests and patches at:
  https://github.com/gnudatalanguage/gdl

(GDL> control_root
% Compiled module: CONTROL_ROOT.
What is your base?
: 20
What is the degree for the root you want?
: 3
What tolerance do you wish to work towards?
: 1e-6
You will need an initial guess, what would it be?
: 2.2
The answer is      2.71442
(GDL>

```

This is the run on the program on the left. First GDL itself has to be executed and then the program called `control_root` is executed in the GDL program. GDL is a freely available software that can run IDL programs. IDL is a common program at NASA/GSFC which is local to the Washington DC area. PV-Wave similar to IDL but is primarily used in California.

```

1 ; Function to figure out roots: A^(1/n)
2 ; Need a tolerance for how close you want the answer to be
3 ; And an initial guess
4 ;
5 ; This is a demo program to show control structures in action while solving
6 ; a math problem (the nth root of A). While most programming languages
7 ; can do that without a program being written this is a method to exemplify
8 ; writing simple program.
9 ;
10 pro control_root
11 print,"What is your base? "
12 read,base
13 print,"What is the degree for the root you want? "
14 read,degree
15 print,"What tolerance do you wish to work towards? "
16 read,tol
17 print,"You will need an initial guess, what would it be? "
18 read,guess
19 answer = 1/degree*((degree-1)*guess+base/guess^(degree-1))
20 while (abs(answer-guess) GE tol) do begin
21     guess = answer
22     answer = 1/degree*((degree-1)*guess+base/guess^(degree-1))
23 endwhile
24 print,"The answer is ",answer
25 return
26 end

```

Comparing the GDL program here to an Octave program on the right shows some differences but a number of similarities. GDL is very much like Octave with similar function and motivations. Note there are no data declarations in either programming language because they are interpreted languages which in general declare on the fly. GDL does have some features that are reminiscent of Fortran (like the *print* and *read*).

```

1 % Function to figure out roots: A^(1/n)
2 % Need a tolerance for how close you want the answer to be
3 % And an initial guess
4 %
5 % This is a demo program to show control structures in action while solving
6 % a math problem (the nth root of A). While most programming languages
7 % can do that without a program being written this is a method to exemplify
8 % writing simple program.
9 %
10 function nul=control_root()
11 base = input("What is your base? ");
12 degree = input("What is the degree for the root you want? ");
13 tol = input("What tolerance do you wish to work towards? ");
14 guess = input("You will need an initial guess, what would it be? ");
15 answer = 1/degree*((degree-1)*guess+base/guess^(degree-1));
16 while (abs(answer-guess) >= tol)
17     guess = answer;
18     answer = 1/degree*((degree-1)*guess+base/guess^(degree-1));
19 endwhile
20 printf("The answer is %g\n",answer);
21 return
22 end
23
24

```

Comparing Octave to the GDL program on the left shows some differences but many more similarities.

```

1 # Function to figure out roots: A^(1/n)
2 # Need a tolerance for how close you want the answer to be
3 # And an initial guess
4 #
5 # This is a demo program to show control structures in action while solving
6 # a math problem (the nth root of A). While most programming languages
7 # can do that without a program being written this is a method to exemplify
8 # writing simple program.
9 # This is for Python 3...Python 2 was sunset on January 1, 2020 but people still use it
10 # Minor changes will be need to convert to Python 2
11 #
12 # >>> import control_root
13 # >>> control_root.control_root()
14 #
15 def control_root():
16     bbase = input("What is your base? ")
17     ddegree = input("What is the degree for the root you want? ")
18     ttol = input("What tolerance do you wish to work towards? ")
19     gguess = input("You will need an initial guess, what would it be? ")
20     base = float(bbase)
21     degree = float(ddegree)
22     tol = float(ttol)
23     guess = float(gguess)
24     answr = 1.0/degree*((degree-1.0)*guess+base/guess**(degree-1.0))
25     while abs(answr-guess) >= tol:
26         guess = answr
27         answr = 1.0/degree*((degree-1.0)*guess+base/guess**(degree-1.0))
28     print ("The answer is ",answr)

```

The program above is written in Python (Python 3 in this case), we have input and output, and a conditional and loop all in one (*while* statement). Because in Python *input* returns a string, the function *float* is used to convert it to a number. Most languages have commands to convert strings to numbers and numbers to strings. Please note how a program is written as most Python tutorials do not spend much time on the *def* command.

```

Python 3.9.4 (default, Apr 6 2021, 12:18:31)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import control_root
>>> control_root.control_root()
What is your base? 20
What is the degree for the root you want? 3
What tolerance do you wish to work towards? 1e-6
You will need an initial guess, what would it be? 2.2
The answer is  2.7144176165949063
>>>

```

This is the run of the program on the left. First Python itself has to be executed and then the program *control_root* has to be imported and then run. Python is a bit more complicated to run programs than either Octave or GDL.

```

1 # Function to figure out roots: A^(1/n)
2 # Need a tolerance for how close you want the answer to be
3 # And an initial guess
4 #
5 # This is a demo program to show control structures in action while solving
6 # a math problem (the nth root of A). While most programming languages
7 # can do that without a program being written this is a method to exemplify
8 # writing simple program.
9 # This is for Python 3...Python 2 was sunset on January 1, 2020 but people still use it
10 # Minor changes will be need to convert to Python 2
11 #
12 # >>> import control_root
13 # >>> control_root.control_root()
14 #
15 def control_root():
16     bbase = input("What is your base? ")
17     ddegree = input("What is the degree for the root you want? ")
18     ttol = input("What tolerance do you wish to work towards? ")
19     gguess = input("You will need an initial guess, what would it be? ")
20     base = float(bbase)
21     degree = float(ddegree)
22     tol = float(ttol)
23     guess = float(gguess)
24     answr = 1.0/degree*((degree-1.0)*guess+base/guess**(degree-1.0))
25     while abs(answr-guess) >= tol:
26         guess = answr
27         answr = 1.0/degree*((degree-1.0)*guess+base/guess**(degree-1.0))
28     print ("The answer is ",answr)

```

Comparing the Python program here to an Octave program on right shows a lot more similarities then differences. One big difference is the indenting of the lines which are required for Python. Python does not have *endwhile*, or ends of any kind, so the way Python tells where the end of a loop is is by the indentation.

```

1 % Function to figure out roots: A^(1/n)
2 % Need a tolerance for how close you want the answer to be
3 % And an initial guess
4 %
5 % This is a demo program to show control structures in action while solving
6 % a math problem (the nth root of A). While most programming languages
7 % can do that without a program being written this is a method to exemplify
8 % writing simple program.
9 %
10 function nml=control_root()
11 base = input("What is your base? ");
12 degree = input("What is the degree for the root you want? ");
13 tol = input("What tolerance do you wish to work towards? ");
14 guess = input("You will need an initial guess, what would it be? ");
15 answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));
16 while (abs(answr-guess) >= tol)
17     guess = answr;
18     answr = 1/degree*((degree-1)*guess+base/guess^(degree-1));
19 end %while
20 printf("The answer is %g\n",answr);
21 return
22 end
23 %
24

```

Comparing Octave to the Python program on the left shows a lot more similarities then differences. Here *input* is used in a similar fashion for both of these programming languages. However, *input* in Octave by default returns a number and for Python it is always a string (hence the float commands). Octave's *input* can be made to produce a string with an additional parameter. Python is more natural for string manipulation; Octave is more natural in the numerical world - hence the popularity of Octave/MATLAB/Scilabs among engineers.

C++

```

1 //
2 // function to figure out roots: A^(1/n)
3 // Need a tolerance for how close you want the answer to be
4 // And an initial guess
5 //
6 // This is a demo program to show control structures in action while solving
7 // a math problem (the nth root of A), while most programming languages
8 // can do that without a program being written this is a method to exemplify
9 // writing simple program.
10 // This is in C++ --- compile with g++-700.gcc
11 //
12 //
13 #include <iostream>
14 #include <stdio.h>
15 #include <cmath>
16 using namespace std;
17
18 int main()
19 {
20     double degree, base;
21     double tol;
22     double guess;
23     double answer;
24     cout << "What is your base? ";
25     cin >> base;
26     cout << "What is the degree for the root you want? ";
27     cin >> degree;
28     cout << "What tolerance do you wish to work towards? ";
29     cin >> tol;
30     cout << "You will need an initial guess, what would it be? ";
31     cin >> guess;
32     answer = 1/degree*(degree-1)*guess+base/pow(guess,degree-1);
33     while (fabs(answer-guess) >= tol)
34     {
35         guess = answer;
36         answer = 1/degree*(degree-1)*guess+base/pow(guess,degree-1);
37     }
38     cout << "The answer is " << answer << "\n";
39 }

```

The program above is written in C++ (not C which is different but very similar since C++ is a direct descendant of C), we have input and output, and a conditional and loop all in one (*while* statement). Note that C++ requires include statements to be able to do a lot of its functions like *fabs*, *cin*, and *cout*. You might also notice that this has data declarations like Fortran.

```

MBP exampleprograms % g++ -c control_root.cpp
MBP exampleprograms % g++ control_root.o -o control_root
MBP exampleprograms % ./control_root
What is your base? 20
What is the degree for the root you want? 3
What tolerance do you wish to work towards? 1e-6
You will need an initial guess, what would it be? 2.2
The answer is 2.71442

```

This is the compiling, linking, and running of the program on the left. Note that *g++* (open source C++) is used here. *gcc* is used for C, not for C++. The executable now can be shared with anyone who has a similar operating system without the user having a compiler. (Note the file ends in *.cpp* which is the convention for C++ files, if this was a C file it would end in *.c*, so look at the ending carefully).

```

1 //
2 // function to figure out roots: A^(1/n)
3 // Need a tolerance for how close you want the answer to be
4 // And an initial guess
5 //
6 // This is a demo program to show control structures in action while solving
7 // a math problem (the nth root of A), while most programming languages
8 // can do that without a program being written this is a method to exemplify
9 // writing simple program.
10 // This is in C++ --- compile with g++-700.gcc
11 //
12 //
13 #include <iostream>
14 #include <stdio.h>
15 #include <cmath>
16 using namespace std;
17
18 int main()
19 {
20     double degree, base;
21     double tol;
22     double guess;
23     double answer;
24     cout << "What is your base? ";
25     cin >> base;
26     cout << "What is the degree for the root you want? ";
27     cin >> degree;
28     cout << "What tolerance do you wish to work towards? ";
29     cin >> tol;
30     cout << "You will need an initial guess, what would it be? ";
31     cin >> guess;
32     answer = 1/degree*(degree-1)*guess+base/pow(guess,degree-1);
33     while (fabs(answer-guess) >= tol)
34     {
35         guess = answer;
36         answer = 1/degree*(degree-1)*guess+base/pow(guess,degree-1);
37     }
38     cout << "The answer is " << answer << "\n";
39 }

```

Comparing the C++ program here to the Fortran program on the right shows more similarities than they did to the interpretive language. C++ has data declarations like Fortran and does not have a combined *print* and *read* statement like Fortran.

```

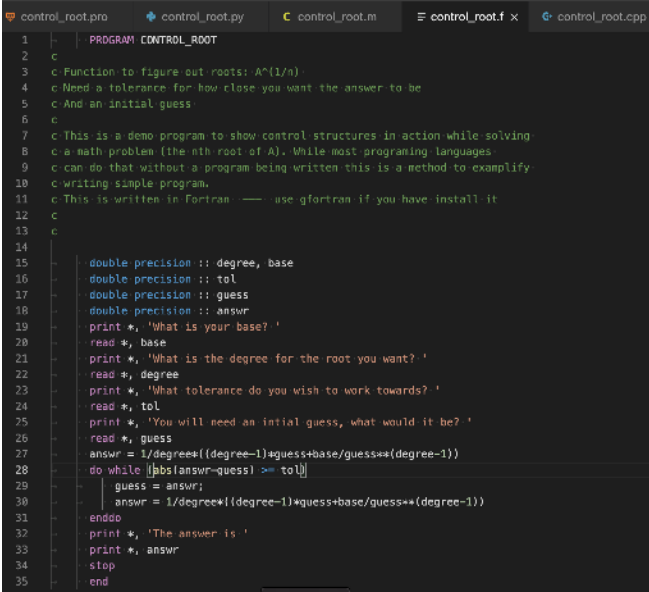
1 PROGRAM CONTROL_ROOT
2 C
3 C Function to figure out roots: A^(1/n)
4 C Need a tolerance for how close you want the answer to be
5 C And an initial guess
6 C
7 C This is a demo program to show control structures in action while solving
8 C a math problem (the nth root of A), while most programming languages
9 C can do that without a program being written this is a method to exemplify
10 C writing simple program.
11 C This is written in Fortran --- use gfortran if you have install it
12 C
13 C
14 C
15 double precision :: degree, base
16 double precision :: tol
17 double precision :: guess
18 double precision :: answer
19 print *, 'What is your base? '
20 read *, base
21 print *, 'What is the degree for the root you want? '
22 read *, degree
23 print *, 'What tolerance do you wish to work towards? '
24 read *, tol
25 print *, 'You will need an initial guess, what would it be? '
26 read *, guess
27 answer = 1/degree*(degree-1)*guess+base/guess**(degree-1)
28 do while (abs(answer-guess) >= tol)
29     guess = answer;
30     answer = 1/degree*((degree-1)*guess+base/guess**(degree-1))
31 enddo
32 print *, 'The answer is '
33 print *, answer
34 stop
35 end

```

Comparing Fortran to the C++ program on the left has some differences, but is essentially the same.

Fortran

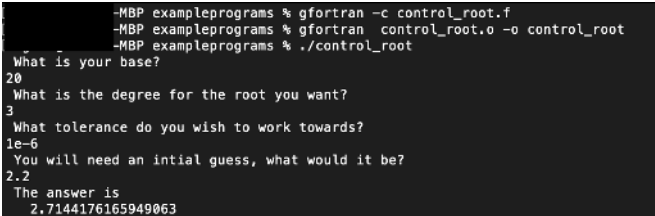
Fortran has already been compared to C++ and Octave so we will not do that here. We will just show the program and the run.



```

1  PROGRAM CONTROL_ROOT
2
3  c Function to figure out roots: A^(1/n)
4  c Need a tolerance for how close you want the answer to be
5  c And an initial guess
6
7  c This is a demo program to show control structures in action while solving
8  c a math problem (the nth root of A). While most programming languages
9  c can do that without a program being written this is a method to exemplify
10 c writing simple program.
11 c This is written in Fortran — use gfortran if you have install it
12 c
13
14
15 double precision :: degree, base
16 double precision :: tol
17 double precision :: guess
18 double precision :: answr
19 print *, 'What is your base? '
20 read *, base
21 print *, 'What is the degree for the root you want? '
22 read *, degree
23 print *, 'What tolerance do you wish to work towards? '
24 read *, tol
25 print *, 'You will need an initial guess, what would it be? '
26 read *, guess
27 answr = 1/degree*((degree-1)*guess+base/guess**(degree-1))
28 do while (abs(answr-guess) >= tol)
29     guess = answr;
30     answr = 1/degree*((degree-1)*guess+base/guess**(degree-1))
31 enddo
32 print *, 'The answer is '
33 print *, answr
34 stop
35 end

```



```

-MBP exampleprograms % gfortran -c control_root.f
-MBP exampleprograms % gfortran control_root.o -o control_root
-MBP exampleprograms % ./control_root
What is your base?
20
What is the degree for the root you want?
3
What tolerance do you wish to work towards?
1e-6
You will need an initial guess, what would it be?
2.2
The answer is
2.7144176165949063

```

The program above is written in Fortran (which is the same as used in the comparisons), we have input and output, and a conditional and loop all in one (*while* statement). Note that Fortran does not include *include* statements like C and C++. A number of Fortran's packages are presumed in linking. This makes Fortran easier for scientists and engineers and one of many reasons Fortran never seems to go away.

Modern Fortran (2008 and beyond) does have a command that is similar to the C/C++ *include* called *use* which is associated with a *module*.

This is the compiling, linking, and running of the program on the left. Note that *gfortran* is used here. The executable now can be shared with anyone who has a similar operating system without the user having to have a compiler.

Comparison of programming languages (morale of previous section)

- Most programming languages are very similar in looks
 - Syntax is the major difference with each programming language having its own unique syntax
 - Program flow is essentially the same in all the five languages introduced above
 - This is true for most languages
 - LISP and LabVIEW are notable exceptions
- The main difference above beyond the syntax is the difference between the interpreted languages and the compiled languages
 - Interpreted languages like MATLAB/Octave/Scilab, Python, and IDL/GDL require another program to run before you can execute your functions
 - That is you cannot execute a function you have made in this languages without running the MATLAB/Octave/Scilab, Python, or IDL/GDL main program
 - Some of these programs do have a way of making a compilable executable but then this makes them a compiled language...
 - Compiled languages like C++ and Fortran do not require another program to execute on any similar machine (with the same OS)
 - Of course you need to compile it first so would need gcc or gfortran (or a similar compiler) before you send your executable to someone else

- The advantage of the compiled executable is you can send your program to people who do not have a compiler

End of chapter statement

The conclusion to this section is that you can learn to program in any language as long as you have the basic ideas of programming. Hence programming in any language is in your grasp. *So you might ask why should I learn how to program in any programming language when company XYZ says they only use Python?* Company XYZ might not keep you on forever so what happens when you go to company ABC and they use [Julia](#)? Knowing how to program is like learning how to fish. Using the modern proverb originating from Anne Isabella Thackeray Ritchie → If you give a person a fish, the person eats for a day; If you teach a person to fish, the person eats for a lifetime. Using the same philosophy, we can conclude that it is better to learn **programming** not learn a specific programming language. At least one author of this text has seen literally hundreds of new programming languages appear since University (of course, Fortran has been there and is still there...so if you want one language, go for that).

In the following chapter we will develop a program from a simple physics problem in order to demonstrate the process of writing a program for engineering (or science). It is limited in scope so as to gently introduce the concept however it will take effort on the students part...

¹Interactive Data Language (IDL) is primarily a local language for PG Community College because of its proximity to NASA/GSFC, other colleges or universities might want to substitute a different regional language. The other languages are global and relatively popular.

²While scripts are good for educational purposes and quick calculations, they are not full fledged program which is the final goal. See parachute person for an example of a full fledged (albeit simple) program

This page titled [3.4: Programming in any language](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [9: Programming in any language](#) is licensed [CC BY-NC-SA 4.0](#).
- [Current page](#) by Julio Terán is licensed [CC BY-ND 4.0](#).

3.5: Mathematical Models

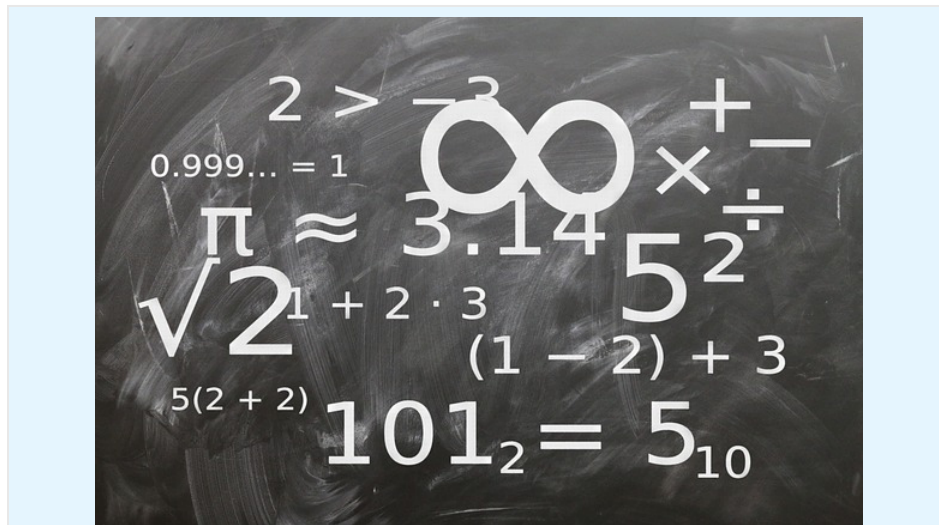


Figure 6.1: Mathematical models are just math ways of talking about natural phenomena.

The last thing we will cover about Excel is how to select a trendline type and what that actually means. It would be better to phrase that as “how to select the correct mathematical model”. **Mathematical modeling** is the process of using math to represent real-world phenomena. The reason the second terminology is better is because it correctly expresses the *thinking* that you must do to select a correct model. Before we jump into the different types of mathematical models (and associated trendlines) that you are expected to learn, we need to have a quick word on *thinking*.

Learning Goals

The emphasis of this chapter will be on thinking. You need to think before you choose a model type. We will also learn how to:

- Explain what a model is and how it is used
- Recognize how to identify a linear, power, or exponential function
- Identify whether an equation represents a linear, power, or exponential model
- Determine the physical meaning and units of parameters of a linear, power, and exponential function.

Think Twice, Model Once

There is an old saying in carpentry that you should measure twice and cut once. The idea is that you can’t un-cut a piece of wood so you better be sure you are cutting it correctly. While this analogy doesn’t perfectly translate to mathematical modeling (you could technically re-run the model), the reality is that once you choose your model and think it is good enough, it is unlikely you will re-scrutinize it.



Figure 6.2: Think twice, model once!

The danger with mathematical modeling is that sometimes your data may *look* like they fit a certain type of model so you select that trendline type and move on. It is easier that way and uses less brain power. Humans are all lazy, and in fact, our laziness has led to some of our greatest inventions! However, your laziness can get the best of you when it comes to mathematical modeling.

You need to *think* about what you would expect the model would be off on your understanding of the underlying physics before you model it. What I am trying to convey is that you *think*, then choose your trendline type later. **Think twice, model once** will be our trendline selection mantra! Now that we have that out of the way, let's look at the different model types that we will consider: linear, power, and exponential.

What is a Mathematical Model?

Engineers use mathematical models **all the time** to represent physical phenomena. For example, in the last chapter we learned about Hooke's Law, $F = kx$ for springs, which is a mathematical model. You may have also heard about Ohm's Law in physics, $V = IR$ for circuits, which is also a mathematical model. There is also Pascal's Law for fluid pressure, $\Delta P = \rho g \Delta H$, Stoke's Law for drag force, $F_d = 6\pi\mu Rv$ and the list goes on and on.

The point is that these mathematical models tell us something about the way things actually occur in the real world. Again, returning to Hooke's Law from the previous chapter, we saw that as we increase the force stretching a spring, the spring will stretch *linearly proportional* to the force applied. That means if we double the force, the distance the spring will stretch will double. We can intuitively understand this because we have all played with springs, and we can also see this reflected in the mathematical equation that describes the phenomena.

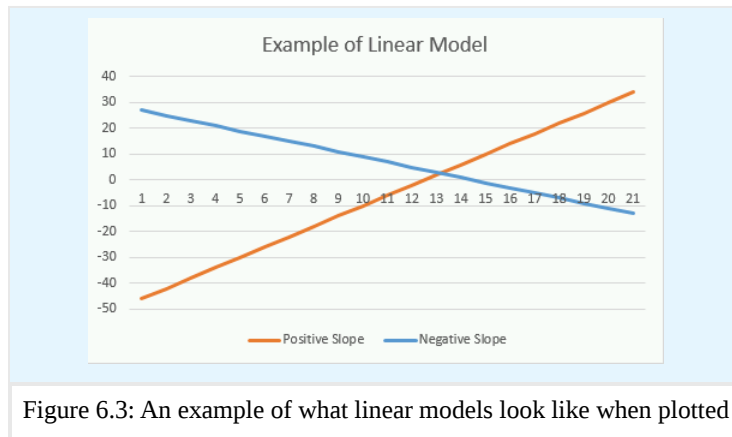
Almost all mathematical models will have the following characteristics:

- They will take the form of an equation $y = \text{something with an } x, m, \text{ and } b$.
- y and x are variables of interest.
- b is a constant (keep in mind sometimes it is 0 or 1 meaning it might be invisible)
- m is the "slope". This will intuitively make sense to you for linear models but you need to keep in mind that the interpretation of slope with the other models will be slightly different.

Linear Models

Linear models are straight lines and take on the form of the equation:

$$y = mx + b$$



Example of Linear Model - Hooke's Law

Let's look at Hooke's Law again:

$$F = kx$$

recall that F is the force deforming the spring, x is the distance it deforms, and k is a spring constant which is a characteristic of the spring (meaning it does *not* change). We can trust that this law is true so let's go ahead and determine what the units of k should be.

Stop & Think

Remember from our units chapter that we can use dimensional analysis to determine the units of k since we know that for this equation to be true, the dimensions on each side of the equal sign must be equal. It follows that the units must be equal on each side as well. I'll be honest, I tell you what the units are for k below. You will only be cheating yourself if you skip this quick brain workout.

Hopefully you actually tried this on your own and found that the SI units for k are N/m . If you google search "buying springs" you can see that this specification is always listed for springs that are for sale. It may be in US units ($lbs/in...gross$) but you get the idea.

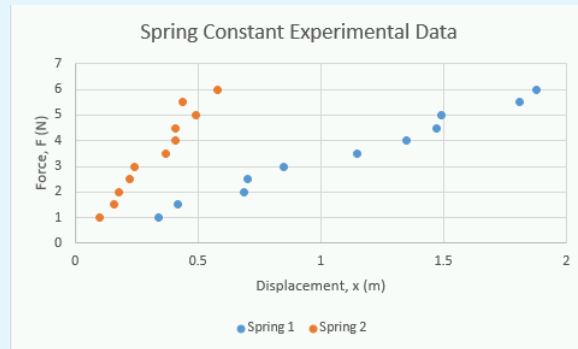


Figure 6.4: Load up the Excel file you saved with this chart.

Now, let's return to the Excel worksheet we were working on in the previous chapter. When we left off, we had a chart with two series plotted on it (we called them Spring 1 and Spring 2).

We know that these are springs and that they should obey Hooke's Law so we can use Excel to add a linear trendline which will effectively mathematically model these equations.

How to Add a Trendline

The process for adding a trendline is identical regardless of the mathematical model being used. Let's see how to add a trendline to our spring data and start with the spring 1 data. The process is also very similar to adding a legend or other chart element so this should be easy for you.

Step 0) Think! What are we mathematically modeling? What type of trendline will we need to use?

Step 1) Click on one of the data points for the spring 1 dataset. You can click on any of them within the graph and you should notice that they are all selected. You can tell they are selected because Excel will put little circles around all the data points in the series (figure 6.5)

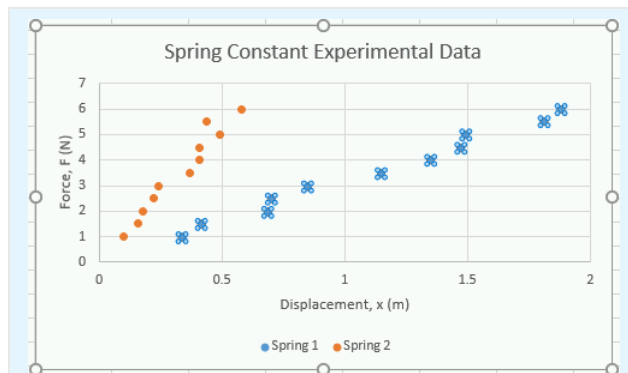
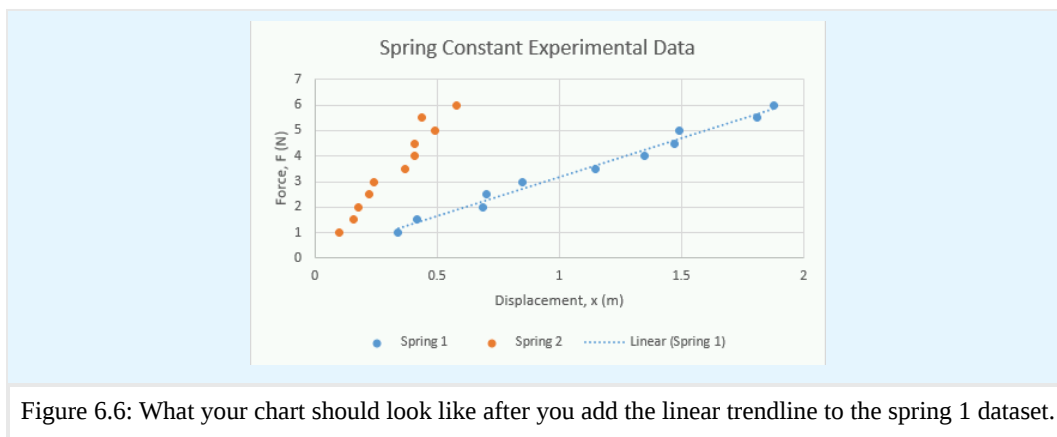


Figure 6.5: Example of Spring 1 data series being selected.

Step 2) Click on our old friend the "Add Chart Element" menu and select the "Trendline" option.

Step 3) In the sub-menu that presents itself, you can choose the type of trendline you want, in this case, it is the “Linear” type. Select the “Linear” option and the trendline should automatically be added to the plot!



Step 4) We have the trendline, but we are still missing the mathematical model that represents the line. To get the model, you need to double-click on the actual trendline itself. That will bring up the “Format Trendline” options (a menu should pop up similar to the one to the left in figure 6.7). It is here that you can change the trendline type if you need to.

Step 5) THINK AGAIN. We know this mathematical model should be linear so we don’t need to change that. But should we set the intercept? Does that make sense in this case? Recall: $F = kx$ and $y = mx + b$. See what I am getting at here? Notice how in Hooke’s Law we are assuming that before the spring has mass added to it, we are expecting it to stretch 0 meters. Does that sound like an intercept?

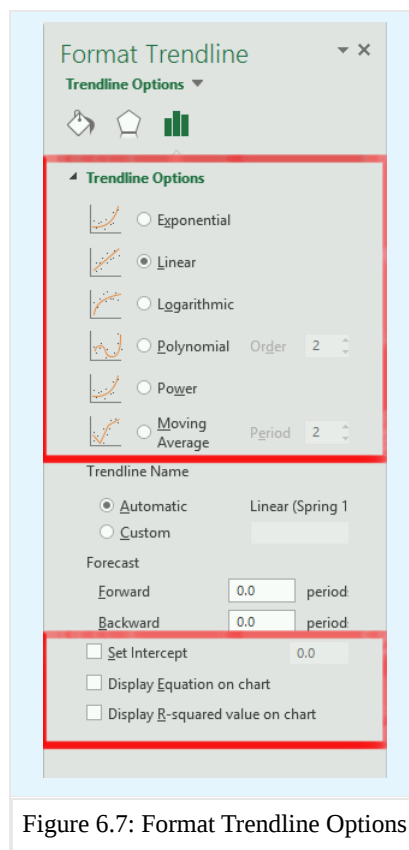


Figure 6.7: Format Trendline Options

Step 6) Yes! Check the “Set Intercept” box and make sure that the value is 0. The $b = 0$ in Hooke’s Law so we should set the intercept = 0. Any deviation from that in our mathematical model will be from measurement errors, it would not be representing actual physical phenomena!

Step 7) Click the “Display Equation on the chart” and “Display R-squared value on chart” options. Then close the “Format Trendline” menu.

Now you can see the mathematical equation displayed on the graph! Notice that the equation is $y = 3.1485x$. The equation is *giving us k , the spring constant!* In this case, we can say that the spring constant is: $k \approx 3.1$

Follow the exact same process for the spring 2 data in the Excel spreadsheet. What is the spring constant for spring 2?

Hint: make sure that you set the y-intercept = 0 so that you get the same answer I did! Also, round to 1 decimal place. It doesn’t make sense to include those non-significant figures. There is no way our measurements had that much precision.

Excel How to add a Trendline aka Ma

Watch on YouTube

Video 1: How to add a trendline to an Excel data series

A quick video recap of this process is provided above in video 1.

Power Model

Power models are swoopy lines and take the form of the equation:

$$y = bx^m$$

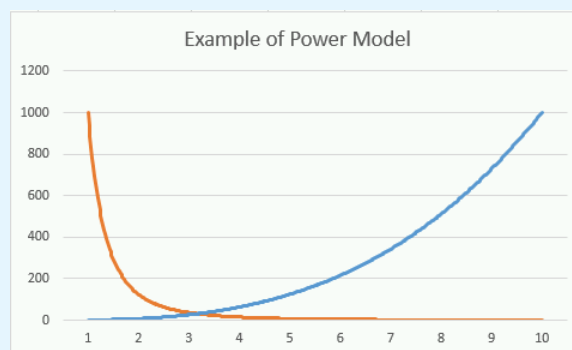


Figure 6.8: Example of two different power models. The blue line has a positive “slope” (m value) and the orange line has a “negative” slope.

There are a few interesting things to note about power models:

- When m is positive, the power model has a value of 0 at $x = 0$
- When m is negative, the power model has a value of ∞ at $x = 0$

Example of a Power Model - Volume of a Sphere

There are many examples of power models in engineering but perhaps the one that you are the most familiar with might be geometric functions. For example, consider the volume of a sphere.

$$V = 4/3\pi r^3$$

in this particular example, V and r are the variables and correspond to the volume and radius of the sphere respectively, $4/3\pi$ is the constant, and 3 (the three in the exponent) is the slope. We can see that if we double the radius, we do not double the volume of the sphere, we increase it by 8 times!

$$\text{If } r = 1, V = 4/3\pi$$

$$\text{If } r = 2, V = 4/3\pi(2^3) = 32/3\pi$$

Fun Fact

This is the reason why insects are small and you don't see human-sized ants running around. Since insects "breathe" by diffusing oxygen through their shell they do not scale up nicely. **Technically, it is a little more complicated** than just straight diffusion but the fact is, geometry limits their size. If an ant was two times bigger, it would have about 8 times more volume which is much more difficult to diffuse oxygen!

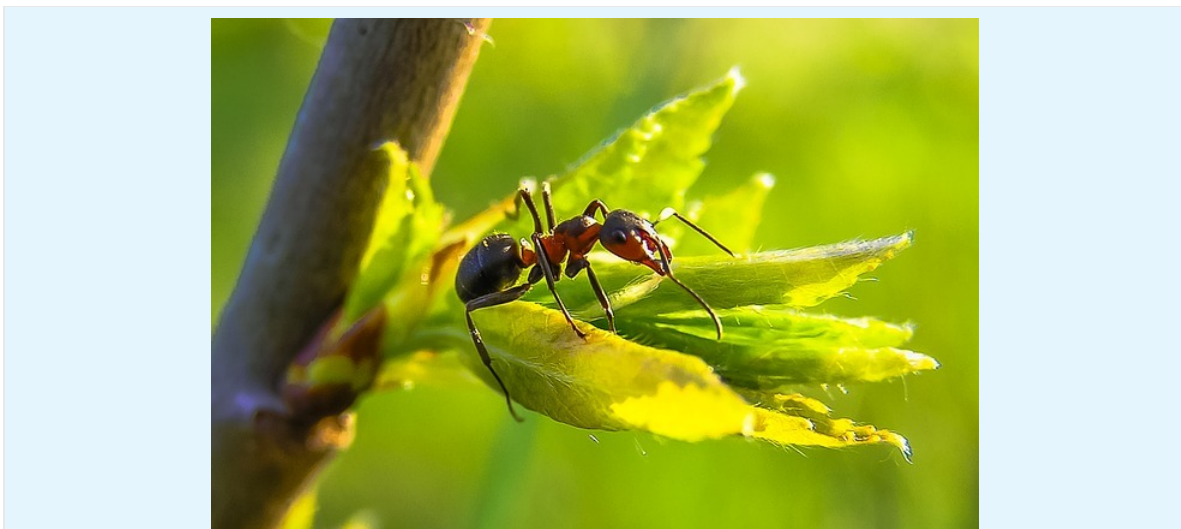


Figure 6.9: Don't worry about him growing to the size of a person anytime soon. Worry about him going extinct.

Identifying the units in this equation is a lot easier. We know that π is a dimensionless number, so the only dimensions are on V (L^3) and on the other r^3 ($L * L * L = L^3$) which checks out.

Other examples of power models in engineering include but are not limited to:

- Kinetic Energy: $KE = 1/2mv^2$
- Ideal gas law: $V = \frac{nRT}{P}$

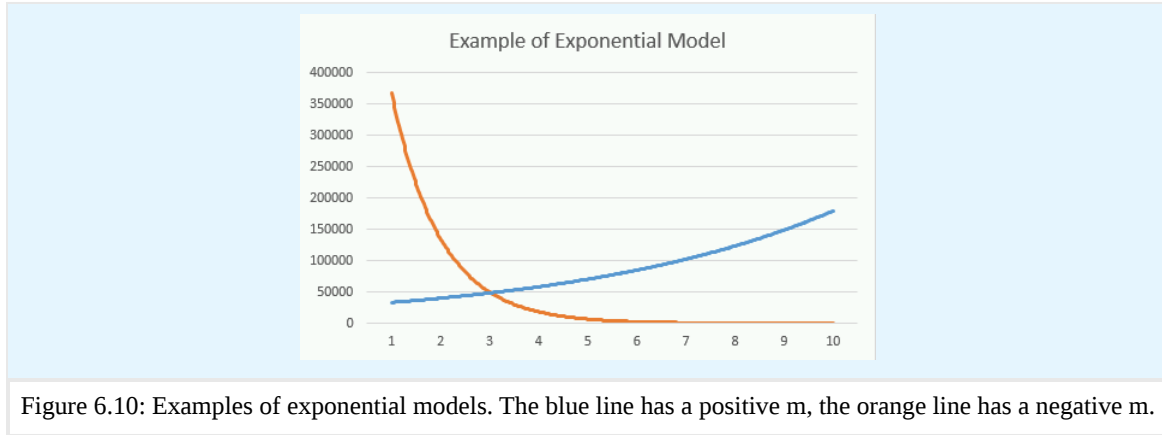
Exponential Model

Exponential models are also swoopy (and hence the reason you need to think to make sure you don't confuse it for a power model!) and take on the form of the equation:

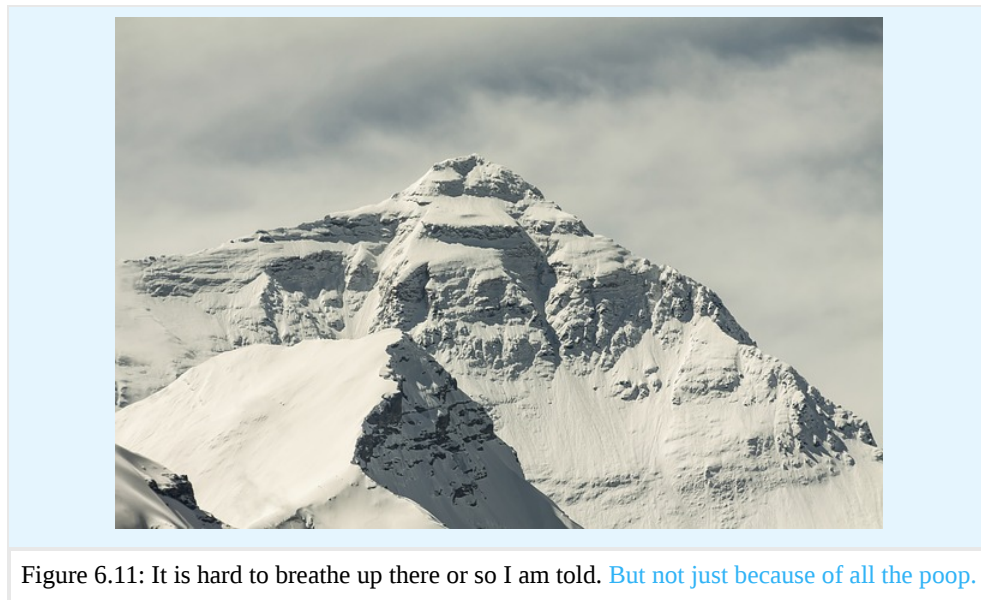
$$y = be^{mx}$$

Keep in mind that the e in this equation refers to **the mathematical constant, the base of the natural logarithm**. There are a couple of other interesting things to note about exponential models:

- The value b is the value at the start of the exponential growth (or decay).
- The m is called the rate of growth.
- When m is positive, the model is asymptotic to 0 for large negative values of x . This scenario is called **exponential growth**.
- Conversely, when m is negative, the model is asymptotic to 0 at large positive values of x . This scenario is called **exponential decay**.



Example of an Exponential Model - Atmospheric Pressure



The “standard” pressure at sea level is 101,325 *Pascals*. The reason standard is in quotes is because realistically it varies with the weather, temperature, etc. but that is the number scientists commonly use to calculate things. Furthermore, it is known that atmospheric pressure, if you could provide existing plans, decreases by about 12% for every 1,000 m that you go up. Intuitively, you can see that this is an example of exponential decay! Furthermore, with a little cleverness, we can create a mathematical model and guess what the pressure should be in Fort Collins, CO (elevation 5,003 ft feet) and on the top of Mount Elbert, the highest peak in Colorado (elevation 14,439 ft or 4,041 m).

The elevation of Fort Collins is listed at 5,003 ft . Take a minute to take out a piece of paper, look at your conversion sheet, and convert this elevation to meters. This is a good brain workout review! Don't skip it!

Looking at our model:

- We know that $b = 101325$ because that is the pressure at sea level in Pa and the b is the value of what we are modeling before the decay.
- We know y is our pressure in Pascals and x is the distance above sea level in meters.
- We know that at $x = 1000$, y is 12% less than at sea level. We can use this to figure out what m is equal to!

12% of 101325 is 12159. That means that $y(1000) = 101325 - 12159 = 89166$. This implies that the pressure at 1000 meters is 89166 Pascals. Make sure you can follow this logic!

Plugging in everything we know to solve m :

$$89166 = 101325e^{1000m}$$

Solving for m :

$$m = \frac{\ln(0.88)}{1000}$$

Now our mathematical model becomes:

$$y = 101325e^{\frac{\ln(0.88)}{1000}x}$$

and we can use this to calculate the expected atmospheric pressure on the top of Mount Elbert!

Atmospheric pressure on the top of Mount Elbert

$$x = 4401$$

$$y = 101325e^{\frac{\ln(0.88)}{1000}4401} = 57727Pa$$

You can validate for yourself with a quick google search that this is a pretty accurate model!

Using the mathematical model we just developed for atmospheric pressure, what is the predicted atmospheric pressure in Fort Collins, Colorado?

End of Chapter Items

This page titled [3.5: Mathematical Models](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 06: Mathematical Models](#) is licensed [CC BY-NC 4.0](#).

CHAPTER OVERVIEW

4: MATLAB

[4.1: Introduction to MATLAB](#)

[4.2: MATLAB Scripting and Arrays](#)

[4.2.1: MATLAB Decisions](#)

[4.3: MATLAB Loops](#)

[4.4: MATLAB Plotting](#)

[4.5: User Defined Functions](#)

This page titled [4: MATLAB](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

4.1: Introduction to MATLAB

MATLAB is a professional software suite and programming language that is commonly used by engineers to analyze data, create complex algorithms (more about these in Chapter 2), and more. MATLAB stands for **matrix laboratory** and is designed to work well with matrices and other types of arrays. MATLAB can help with digital signal processing, control systems, computer vision, and the list goes on. It is stable, robust, relatively fast, well optimized, and can run on Windows, Mac, Linux, and even on Embedded Systems. MATLAB is a pretty amazing bit of software that is well maintained, regularly updated, and well supported.

But...

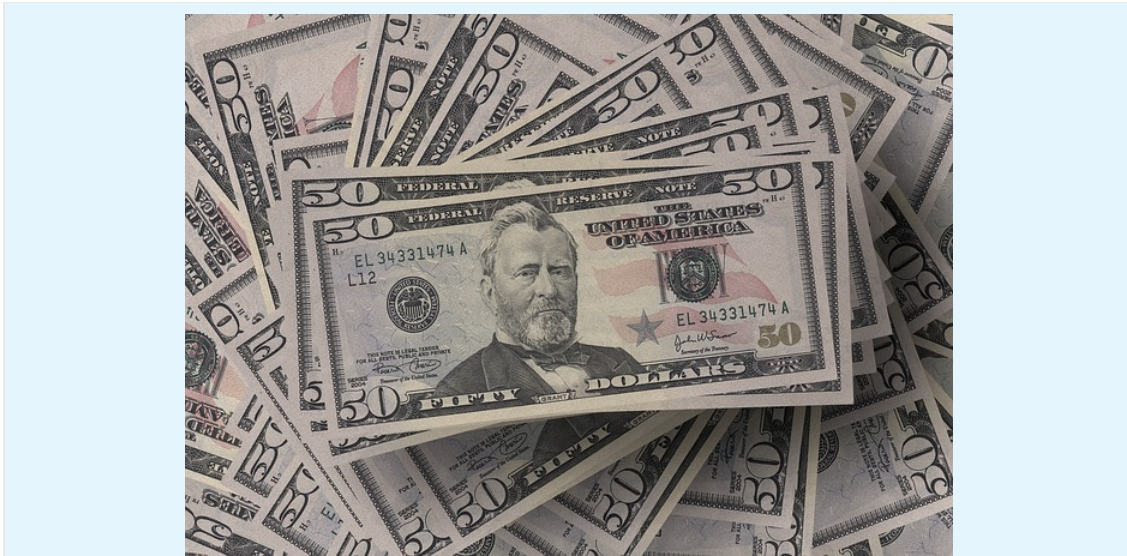


Figure 7.1: MATLAB is so expensive it should be featured in rap videos as “bling”. Dolla Dolla bills, y’all.

MATLAB is *pricey*. [Currently, a single standard license is \\$2150](#). Obviously, any piece of software that can cost \$2150 *per user* to a company must be worth learning about but I would understand the hesitation to purchase that as a student.

Luckily, if you are a student you can sometimes get MATLAB for the low, low price of \$0. Some universities pay for their engineering students to have MATLAB licenses! Inquire with your University’s IT departments to see if you can download MATLAB for free on your personal computer. If your university does not have a site license, [you can purchase a student license for \\$49](#), which isn’t that bad. Since this book is going to use MATLAB in every chapter, you will need to figure out a way to have access to a MATLAB installation.

Learning Goals

In this chapter, we will explore the basics of MATLAB which will culminate in learning how to use MATLAB as an overpriced calculator. That means learning:

- Why you should bother to learn MATLAB instead of <insert_programming_language_here>.
- What are the different components of the MATLAB GUI are.
- How your computer stores files and why that is important to MATLAB.
- How to use MATLAB as a glorified calculator.
- How to use variables to store information and simplify your calculations.

Why MATLAB instead of <insert_programming_language_here>?

At this point in your life, you have probably been exposed to different software, you might even have programming experience, and it is even possible that you have a preference for a specific programming language. So why MATLAB instead of Python? Instead of C++? Instead of Go? (Clearly, the list could go on and on).

The reason this book concentrates on MATLAB is that as of the publication of this book, **MATLAB is an extremely popular programming suite that engineers are expected to know how to use**. MATLAB is popular because it is a general-purpose language that lends itself to productivity. MATLAB is worth knowing because...

- It is easy to express computational mathematics. (You will see this shortly later in the chapter!)
- MATLAB is *specifically designed for engineers* which means the documentation and functions are easy to use and remember.
- The environment is tuned for iterative workflows and rapid prototyping.
- There are a TON of toolboxes which means that a lot of code is already available for you to use. You do not need to reinvent the wheel.
- The MATLAB debugging tool is extremely powerful and easy to use.
- MATLAB is *fast*. It is optimized and can solve big problems quickly.

The list goes on and on but MATLAB is expensive for a reason. The reason is that it is a powerful, easy-to-use software development suite designed for rapid prototyping. Furthermore it was designed specifically for engineers which means the documentation is excellent and tuned toward engineers and the functions available were written with engineers in mind.

MATLAB GUI

When you are first learning MATLAB it may seem easy to ignore the MATLAB graphical user interface (GUI) and just focus on the command window. It is important to learn what each section of the MATLAB GUI does, and how you can use it to your advantage. When you first open MATLAB, you should see something very similar to the default GUI shown below.

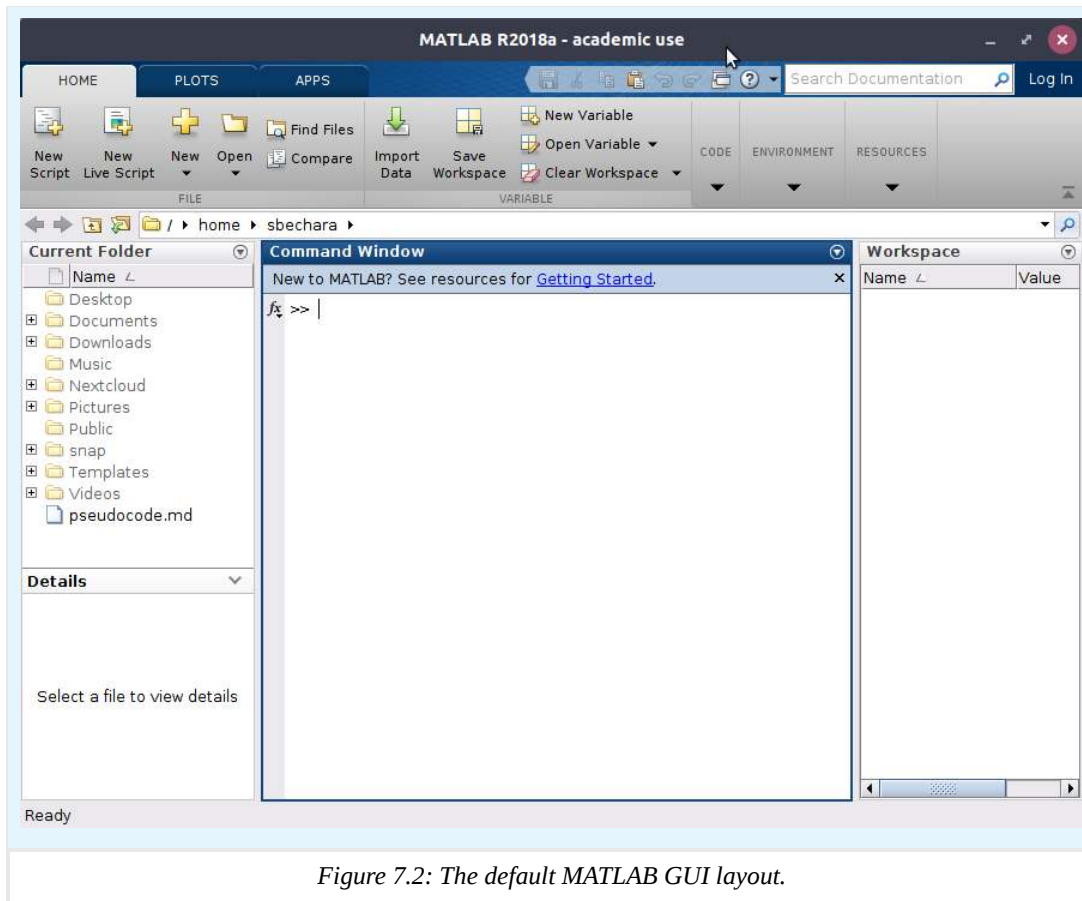
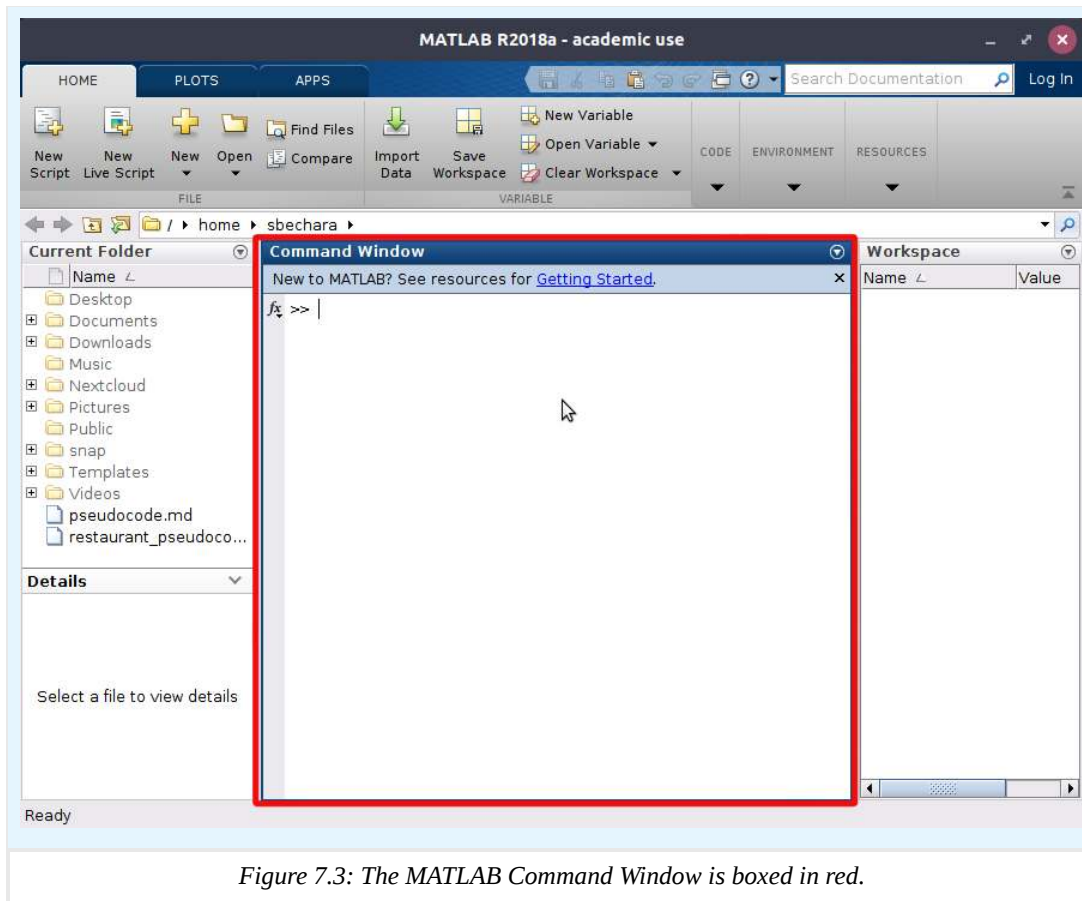


Figure 7.2: The default MATLAB GUI layout.

Let's dive into some of the areas of this program and explain what they do, and how you will use them. In each of the sub-sections below, the area of interest will be boxed in red. It is highly recommended that you open up a copy of MATLAB and click around as you are learning about each part of the workspace. Remember to work out your brain!

Command Window



The command window provides flexibility to MATLAB and how users interact with your programs. It is the place where you can issue a single line of code to be executed by MATLAB. *You can issue any command that is a valid line of MATLAB code.* The command window is a unique feature of MATLAB and is an invaluable tool for debugging programs, testing functions, making quick calculations, and more.

In your career, you will often use the Command Window for quick calculations or to check your work (in essence turning it into a very expensive calculator). For example, lets say that you are working on a problem and need to convert 15.23 *degrees* to *radians*. You remember the conversion factor is $\frac{\pi \text{ radians}}{180 \text{ degrees}}$ and when you calculate it by hand you get 15.23 *degrees* = 0.27 *radians*. You want to check your answer in MATLAB and you remember that MATLAB has a built-in function to automatically convert a number from degrees to radians (`deg2rad()`). So you quickly type the following in the command window (see Figure 7.4). Voila! You were right. Don't panic yet! I don't expect you to know the function `deg2rad()` yet, I am just showing an example of how to use the command window.

```

1 % Here is an interactive code box that functions identically to the MATLAB command window.
2 % These code boxes will be used throughout the upcoming chapters as a way to run code within the textbo
3 % Octave is a free programming language that for your purposes will work the same as MATLAB
4 % Hit run to see the output of this degrees to radian function
5
6 deg2rad(15.23)

```



Figure 7.4: Example of how to use the Command Window.

Current Folder

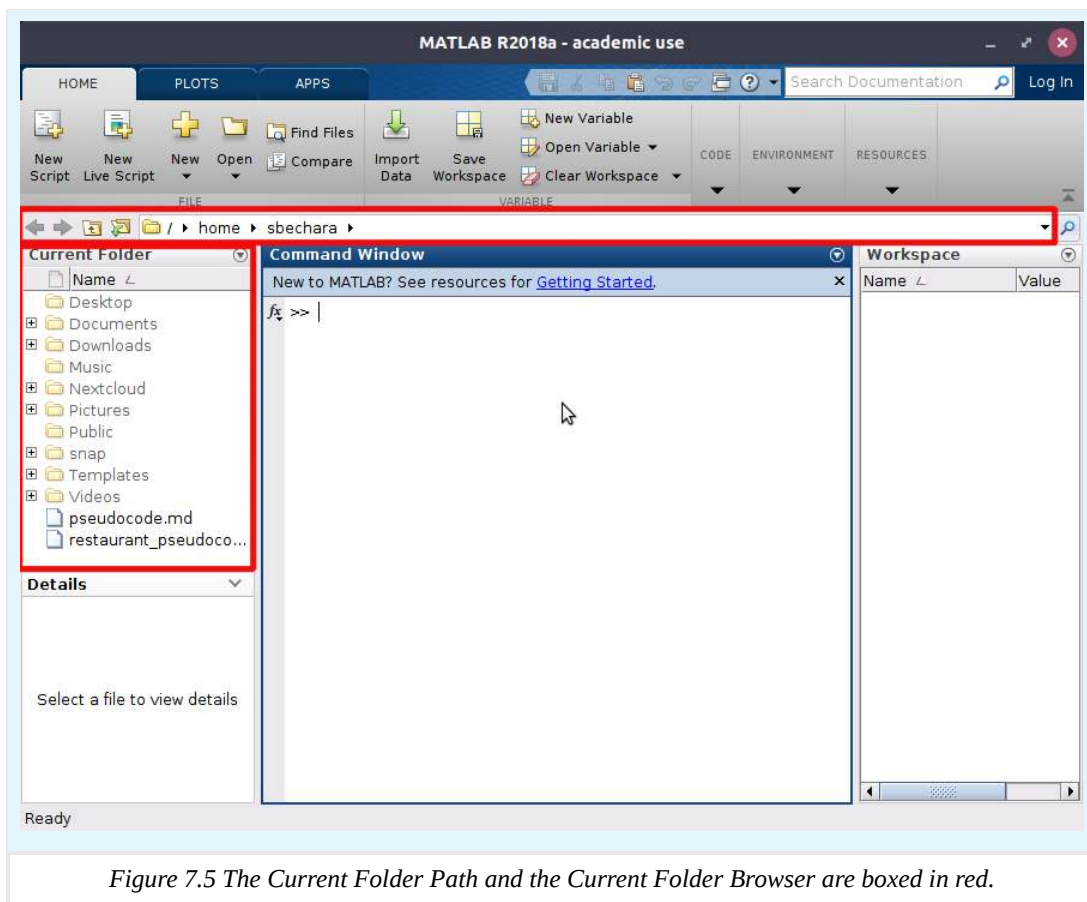


Figure 7.5 The Current Folder Path and the Current Folder Browser are boxed in red.

It is very important to organize your files and folders when working with MATLAB. One of the most common problems students have when starting to learn MATLAB is improper file management. They start storing all of their script files, functions, and algorithms inside the same folder. While this works for 2 or 3 files when you have 200 or 300 in the same folder, things get messy. Today is the day to get your digital life organized! I suggest creating a new folder in your documents folder that is named after the name of your course. So if you are taking MECH103: Intro to Mechanical Engineering, name your folder `mech103` (or something else that makes sense and makes you happy). Then add folders into *that* folder as appropriate. For example, it might be a

good idea to create a folder called `assignments` that you use to store your assignments into, etc. It is assumed that you understand how and where to store things on a computer.

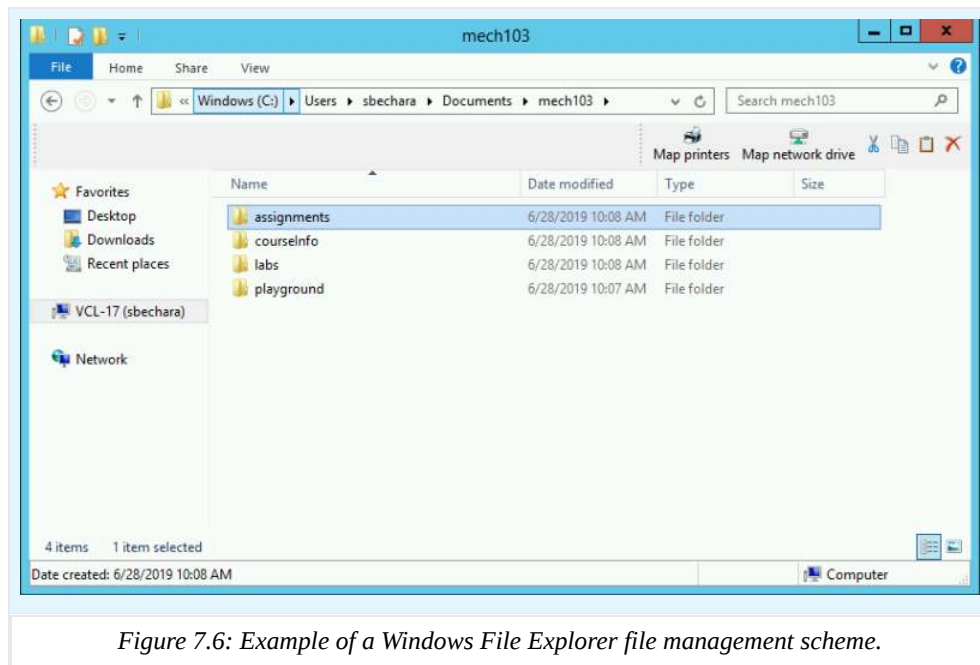


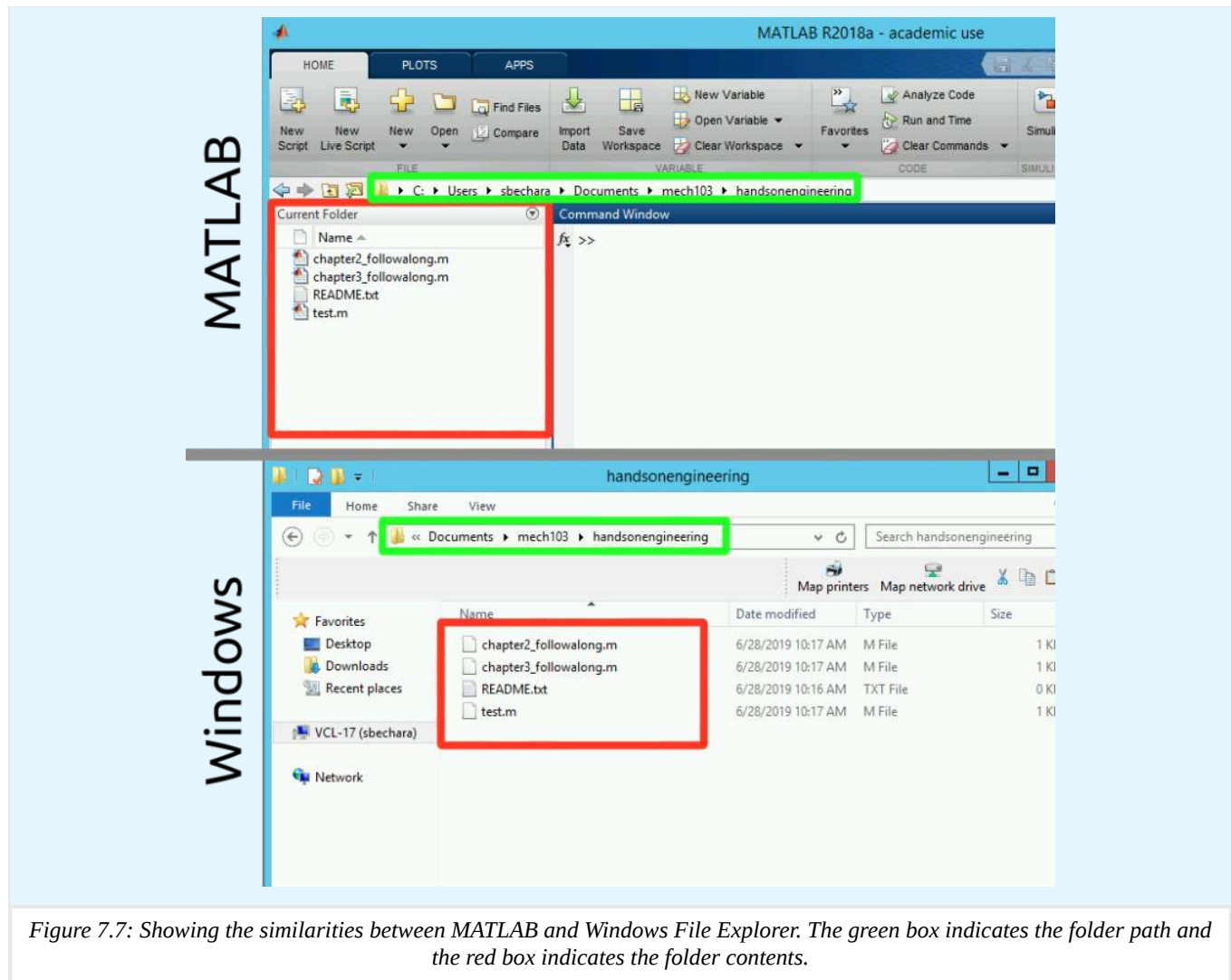
Figure 7.6: Example of a Windows File Explorer file management scheme.

You do not need to be an IT Professional to understand MATLAB. But a critical skill is understanding file structures and directories. For example, you should know that the “path” to the folder shown in figure 7.6 above is `C:\Users\sbechara\Documents\mech103`. If you need a quick refresher on Windows File Structures, try watching the following video to catch up. I personally use GNU/Linux operating systems so when you see screenshots of MATLAB you may notice that the path is a little different (UNIX-like systems such as macOS and Linux use `/` instead of `\` and do not have drive letters like `C:` or `D:`).



Video 7.1: Windows File Structures Review Video

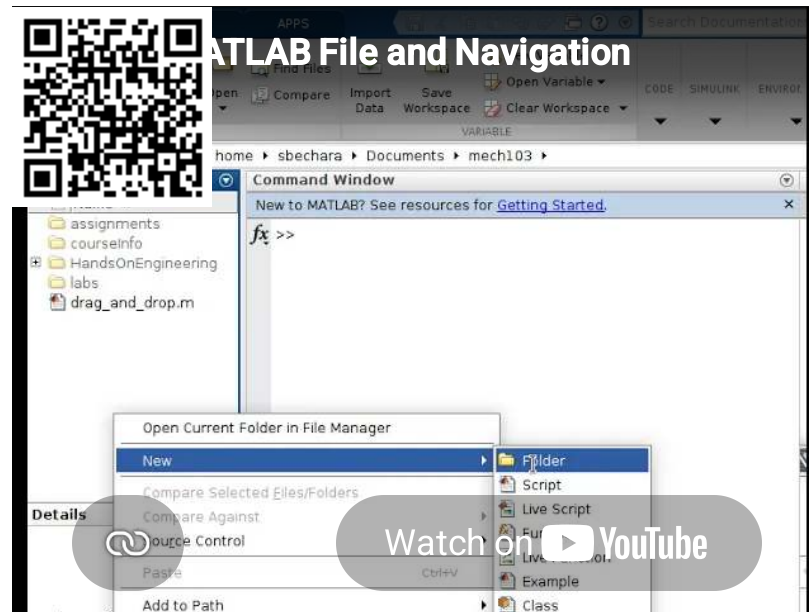
To illustrate the similarities between MATLAB's current folder browser and the current folder path to Windows Explorer, see figure 7.7 below. In figure 7.7, **the green boxes indicate the folder path (also sometimes called the current working directory)** (note: in figure 7.7 Windows truncated and doesn't show the full path, to see the full path click the << to the left of Documents) which in this case is `C:\Users\sbechara\Documents\mech103\handsonengineering`. The **red boxes indicate the current folder contents**, since the folder is the same we would expect both the Windows File Explorer and MATLAB File Explorer to contain the same files, and they do! The only difference is that MATLAB puts different cute icons to help you quickly understand what your files are. Isn't that nice?



The next thing to learn is how to navigate your computer's file system inside MATLAB. **You do it the same way that you navigate Windows File Explorer in Windows or Finder in macOS.** The problem here is that there are a ton of options. I encourage you to take a small brain workout and try each of these in MATLAB on your own.

- You can click inside the path and type in a new path
- You can double-click the folders displayed in the current folder
- You can click on the little + symbols to show sub-directories and their contents
- You can right-click inside the current folder window and add new folders and files
- More?

I created a video showing me navigating a file system. See Video 7.2 below for an example. Again, **do not let this be a substitute for actually trying it on your own!**

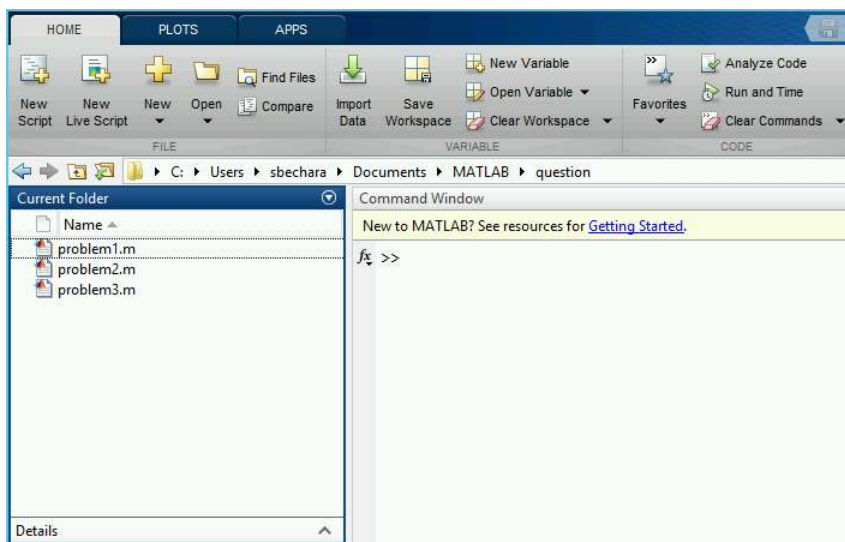


Video 7.2: Examples of MATLAB file and folder organization and manipulation capabilities.

Examples of MATLAB file and folder organization and manipulation capabilities. Can be seen in the video above

So the main thing to remember is that the “Current Folder” window in the MATLAB workspace is showing the files in the folder MATLAB is currently working out of. The way to navigate files and folders within MATLAB should be familiar to any Windows, macOS, or Linux user. The keys are to be able to know where you are and what is in the folder you are currently in.

Look at the MATLAB screenshot taken on a Windows computer. What is the user’s current path (also called the current working directory) based on this image?



Workspace

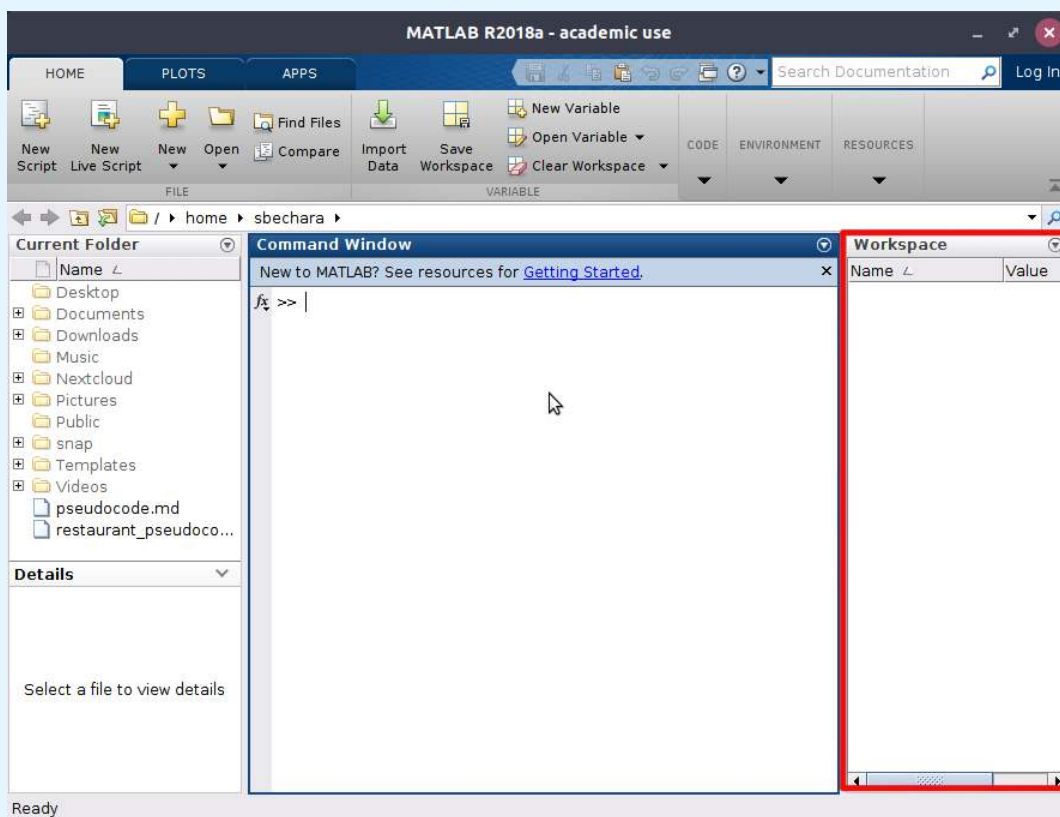


Figure 7.8: The MATLAB Workspace is boxed in red. Currently, it is empty indicating that there are no workspace variables available to the user.

The area labeled “Workspace” and boxed in red above in figure 7.8, is the place where MATLAB lets the user know what variables are loaded and available to the user. We will learn about variables below, but for now, you can think of them as a string of letters that can be used to store a number.

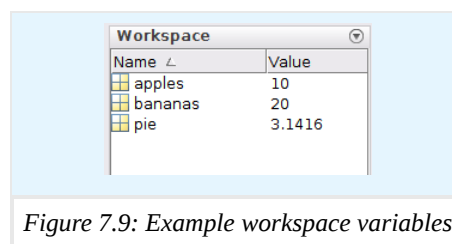


Figure 7.9: Example workspace variables

In figure 7.9 to the left, notice that there are workspace variables. In this case, there are 3: “apples” has the value 10 stored in it, “bananas” has the value 20 stored in it, and “pie” has pi stored in it.

Again, you will learn later how to create your own variables and use them, but for now, it is just important that you understand how MATLAB displays the information to you about what variables are currently defined and what numbers they contain.

MATLAB Banner



Figure 7.10: The MATLAB Banner

The MATLAB banner is the part of the GUI that sits on top of everything else. There are multiple tabs (that will actually change depending on what you are doing) and there are a bunch of buttons with a little descriptive text. Almost all of the buttons are labeled in such a way that it should be pretty obvious as to their function.

For example, the “New Variable” button creates a new variable in the workspace. The “Preferences” button opens up a modal dialogue box that contains a whole bunch of options that you can change in MATLAB.

Take some time to familiarize yourself with the banner and the buttons located on it (especially the buttons in the “Home” tab). In the next chapters, I will refer to the buttons on the banner so you must know where to look.

Try It!

Take a second or two to play around with the buttons on the “Home” tab of the banner. You can’t really break anything and it is good to explore them to see what happens. Also, take a second to click through the other available tabs to see what buttons are available on those. A lot of what we will be learning throughout the rest of this book will rely on your ability to find the correct button on the banner so don’t skip this step!

Quick Recap

If you were paying attention, working out your brain muscles, and following along, hopefully now you know:

- What the command window is and what it is used for.
- What the current folder path and current folder browser are, how to navigate your file system using the current folder browser, and know how to add and remove files from your file system using MATLAB.
- What the Workspace area is for, how to interpret what variables are currently stored in MATLAB, and get a quick idea about the values they contain.

If any of the above three bullet points seem fuzzy to you, please take some time to re-read those sections or seek help elsewhere. From this point forward, it will be assumed that you understand these three areas of the MATLAB GUI and what they are used for.

MATLAB as a Glorified Calculator

Even though we spent the preceding section praising MATLAB for its powerful capabilities we are going to spend the rest of this chapter learning how to use it as a calculator. It is going to seem that this is boring and easy, however, there are key concepts that are easy to miss! To best learn, I recommend that you open MATLAB and practice these concepts in the MATLAB command window.

Arithmetic Operators (for scalars)

To use MATLAB as a calculator, we first need to learn what symbols are reserved for use as arithmetic operators. Keep in mind this section only applies to arithmetic performed on **scalars** (A quantity (such as mass or time) that has a magnitude describable by a real number and no direction. In other words, it is NOT a vector!). Future chapters will cover mathematical operations with vectors. Most of the following operators should seem pretty obvious to you.

Symbol	Role
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

Table 7.1: The basic MATLAB mathematical operators.

Now that we know the symbols, we can execute basic mathematical operations in MATLAB!

```
1 % Here is a code box to use for this question if you would like
```

run

run all

restart

restart & run all

Important Symbols / Commands

As we continue throughout this book, we will be adding to the list of MATLAB commands and reserved symbols. For now, these symbols and commands are the ones that I think are the most important for you to learn.

Symbol / Command	Description
%	This symbol signifies to MATLAB that the line is a "comment". Comments are lines of code that are not executed by MATLAB, but instead are ignored. They will come in handy when we begin to write script files and will make our code more readable by humans.
;	The semicolon symbol is used to repress output to the command window and is included at the end of a line of MATLAB code. This is best to just see in action. Try typing 4+4 in the MATLAB command window then try typing 4+4;
clc	When entered, this command will clear the command window. It does not erase variables from the workspace (a concept we will cover later) it just cleans things up. Again, try typing this into the MATLAB command window to see in action
clear	This will erase all of the variables in the workspace. To see this in action, pay attention to your workspace. If you have been playing around a bit in MATLAB it is likely you have an "ans" variable in your workspace. Notice when you type in clear and hit enter, that variable dissappears.

Table 7.2: Other important MATLAB commands and symbols.

Order of Precedence

Analogous to what you learned in previous mathematics courses about **order of operations** (A lot of you probably learned an acronym like PEMDAS: Parenthesis, Exponents, Multiplication/Division, Addition/Subtraction) MATLAB has a built-in **order of precedence**. The concept is the same, the order of precedence is built into MATLAB and specifies the order in which mathematical operations will be performed.

As we learn new operators (specifically logical and relational) we will be updating this list. For now, the MATLAB Order of Precedence is as follows.

1. Parenthesis
2. Exponentiation
3. Multiplication / Division
4. Addition / Subtraction

Try It!

For example, consider if MATLAB were to execute the following two mathematical operations:

```
>> 4 + 6 * 2
```

and

```
>> (4 + 6) * 2
```

In the first instance, based on the order of precedence, the result would be `16` . Why? MATLAB will evaluate the `6 * 2` portion of the operation first, as it is higher in order of precedence than addition.

In the second instance, the result would be `20` . In this case, the operations in the parentheses will be evaluated first `(4 + 6)` and then the result of that will be multiplied by `2` .

Variable Basics

To complete our lesson on how to use MATLAB as a calculator we need to discuss variables. Luckily, assigning variables in MATLAB is very easy and MATLAB is very forgiving with its variable naming rules. To assign a variable, you simply use the `=` operator. (Note: you don't need to type in the `>>` . From now on I will put the `>>` in when you should type a command into the command window. Notice how the command window prompt in MATLAB is `>>` .)

```
>> csu = 1881;
```

In this case, the variable named `csu` will be assigned the value `1881` (recall that the semicolon simply suppresses the output to the command window. Try typing the example into MATLAB yourself with and without the semicolon to see it in action!). At this point, you should notice that a new variable `csu` has been added to the workspace. Now we can use the variable `csu` in any mathematical operation.

```
>> csu + 19
```

You should notice that without the semicolon, `1900` is output to the command window. Sure enough, that is `1881 + 19` ! The other thing to notice is that if you do not specify a variable to store an operation in, MATLAB will default to storing the value in a variable named `ans` . You can see this in the workspace. The `ans` variable is OK, but it is not a safe place to store information. MATLAB will overwrite the `ans` variable every time an operation is performed that does not include an assignment. Let's re-try that last operation and this time assign the resultant to a new variable called `total` .

```
>> total = csu + 19
```

When you try this, you should notice a NEW variable named `total` added to the workspace and that it contains the value `1900` . If you have been following along, you should have three variables in your workspace; `csu` , `ans` , and `total` .

The last thing we should mention about variables is the variable naming rules:

- MATLAB variables **must start with a letter**
- After the first letter, other letters, digits, or underscores can follow. No other characters can be used
- Variables are case sensitive meaning the variable `RAMS` and `rams` are NOT the same
- This is just a guideline but your variable names should be descriptive of the quantity that they are storing

Important Functions

MATLAB includes *thousands* of built-in functions that can do calculations for you. That is actually one of the reasons why it is so expensive (it costs a lot of money to pay programmers to make those functions for you!). Functions in MATLAB work very similar to functions in Excel in that you give them some input (typically a number or variable that has a number stored in it) and they return a value based on some calculation. For example, if you wanted to calculate the square root of 16, you could type the following into the command window:

```
>> sqrt(16)
```

In this case, the function is `sqrt()` and the input is `16` . When you hit enter you should see that MATLAB spits out:

```
ans =
```

```
4
```

Which is the square root of 16 (remember to leave off the semi-colon when we want to see the answer displayed in the command window). We could have also stored 16 in a variable and asked for the square root of the variable.

```
>> number = 16;
```

```
>> sqrt(number)
```

Will work the exact same way.

```
1 % Here is a code box to try the sqrt() function
2
3 % Try entering different numbers and mathematical expressions and see what happens
4
5 number = 16;
6
7 sqrt(number)
```

run

run all

restart

restart & run all

We won't be learning about every MATLAB built-in function available, but here are a few that you may find useful right away. For your brain workout, spend a few minutes looking over table 7.3 below and practicing using those functions. I suggest trying a combination of numbers and variables as input arguments to the function.

Function	Explanation
<code>sqrt(var)</code>	Calculates the square root of <code>var</code> .
<code>cosd(var)</code> / <code>cos(var)</code>	Calculates the cosine of <code>var</code> in degrees / radians respectively.
<code>sind(var)</code> / <code>sin(var)</code>	Calculates the sine of <code>var</code> in degrees / radians respectively.
<code>tand(var)</code> / <code>tan(var)</code>	Calculates the tangent of <code>var</code> in degrees / radians respectively.
<code>abs(var)</code>	Returns the absolute value of <code>var</code> .
<code>log(var)</code> / <code>log10(var)</code>	Calculates the natural logarithm / base 10 logarithm of <code>var</code> respectively.
<code>round(var)</code>	Rounds <code>var</code> to the nearest integer value.
<code>ceil(var)</code>	Rounds <code>var</code> up to the nearest integer value.
<code>floor(var)</code>	Rounds <code>var</code> down to the nearest integer value.
<code>rem(x,y)</code>	Returns the remainder after division of <code>x</code> by <code>y</code> (<code>x</code> is the dividend and <code>y</code> is the divisor).

Table 7.3: Some important built-in functions that you should be able to use.

```

1 % Question 7.5: Degrees or Radians
2 % Use this code box to compare the value for sin of 360 radians versus 360 degrees

```

run

run all

restart

restart & run all

Getting Help

As you can probably already tell, there is *a lot* to learn when you are studying MATLAB. *It is not likely that you will remember everything you read about in this book.* While the book is and will continue to be a resource for you, I thought it would be useful to point out two tools that can help you when you feel like you are stuck. I recommend that you attempt to seek out help in the following order:

1. First try: MATLAB Built-In `help`
2. If that doesn't work, try: Google (*dangerous*)

Be honest with yourself and remember that we are here to work out our brains! It is OK to seek out help from external resources but you need to make sure that you are *learning* the material.

MATLAB Built-In Help

The MATLAB help documentation is very good and is a good place to start when you are stuck. The only problem is it will only help you if you *know* what function or command you need help with. For example, if you are confused about how to use the `cosd()` function you can type in:

```
>> help cosd
```

```
1 % While Octave is slightly different then MATLAB, both have help text
2 % Try looking at Octave's help text for the sind() function
3
4 help sind
```

run

run all

restart

restart & run all

and MATLAB will display a descriptive help text explaining what the function is and how it works. **This is the best way to get help when you are first learning MATLAB.** It forces you to think about what you need help with specifically and it doesn't answer your problem *exactly*. That is a good thing when you are practicing. It is OK to get a little spot when you are working out, but you don't want your spotter pulling the bar up for you because then you wouldn't be working out your muscles! In fact, a good brain workout would be to go into the list of functions above in table 7.3 and read through the help text on each of those functions. Some of the text may be gibberish to you, but it is a good idea to start practicing reading technical documentation now.

Using Google for Help

If you are stuck and you have read through MATLAB help text and you still need a little extra help, you can usually use google to search for and quickly find an answer to your problem. However, I caution you to do this only in extreme circumstances! This is the equivalent of having your spotter lift the bar for you above. If you are totally maxed out and can't figure it out, it is OK, but you need to go back a little later and try it again for yourself to make sure you learn how to do it on your own.

Another good resource is the [MATLAB subreddit](#). If you are not familiar, Reddit is a forum website where users can submit questions and get answers from a community. I actually frequently respond to posts on the MATLAB subreddit so if you post a question there, I might be the one answering it for you!

In the end, it is important to remember, that the goal of this book is to work out your brain. I will trust that you will use the internet with discretion, understanding that learning the material is more important than “getting the answer”.

Try It!

At this point, the best thing you can do is play around with MATLAB as a calculator. Do some arithmetic operations, assign some variables, use the semi-colon, omit the semi-colon, try typing in clear to see what happens, etc. DO NOT SKIP THE PLAYING STEP. You can read about these concepts, but if you want to learn you need to see them in practice. The best way to do that? PLAY!

```
1 % Use this code box to mess around with some MATLAB skills you have learned in this textbook
```

run

run all

restart

restart & run all

This page titled [4.1: Introduction to MATLAB](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 07: Introduction to MATLAB](#) is licensed [CC BY-NC 4.0](#).

4.2: MATLAB Scripting and Arrays

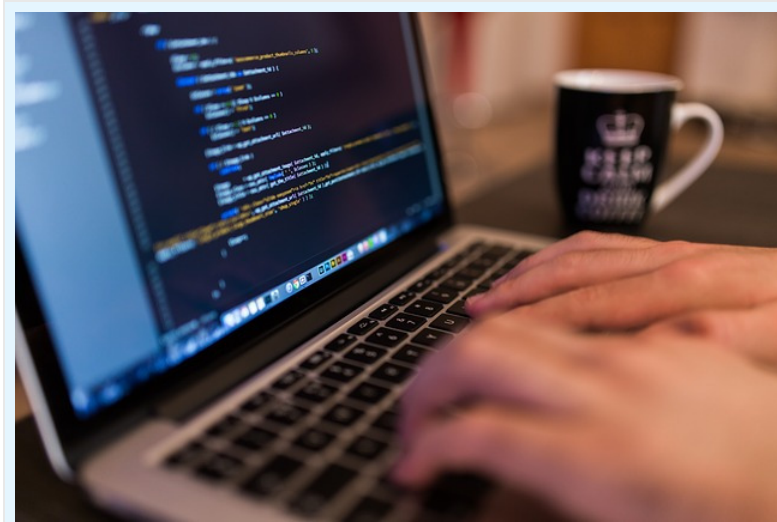


Figure 8.1: Type, type, type.

To start this chapter I would like you to consider a situation where you need to perform the same *calculation* with *different numbers* multiple times. For example, let's pretend that you are preparing for a final exam, and you would like to know how different final exam grades will impact your overall average. You have already taken midterm 1, midterm 2, and midterm 3 so those grades are locked. You recall that to calculate the average all you have to do is sum your values and divide the sum by the number of entries. For you math folks, the formula is:

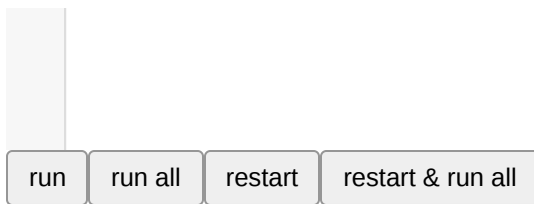
$$average = \frac{1}{n} \sum_{i=1}^n a_i \quad (4.2.1)$$

Your MATLAB code might look something like this. Feel free to type this into the MATLAB command window yourself, it is good practice).

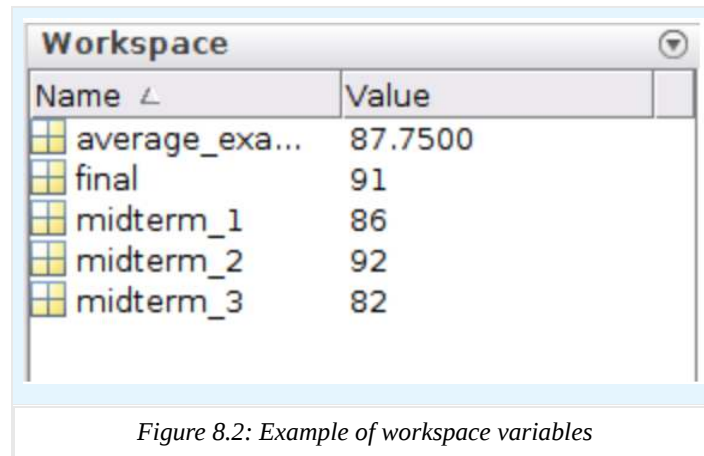
Try It!

```

1 % Try It 8.1
2
3 midterm_1 = 86;
4
5 midterm_2 = 92;
6
7 midterm_3 = 82;
8
9 final = 91;
10
11 average_exam_score = (midterm_1 + midterm_2 + midterm_3 + final) / 4;
```



Note: there are actually even better and more clever ways of performing this exact same calculation in MATLAB but for now, let's stick with this example.

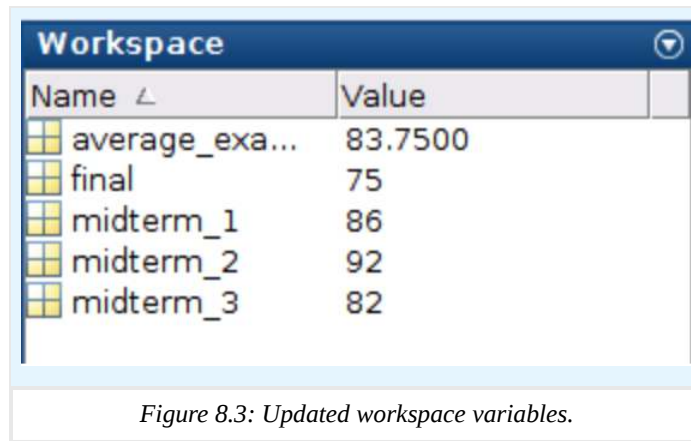


Viola! You can see that your average exam score would be 87.75 . Now let's consider how a 75 on the final would affect the average exam score. We can see that the variables for the var scores are still present in the workspace (Figure 8.2 above). All we need to do is overwrite the final exam score so that it is 75 , then re-run the average exam score calculation.

```
>> final = 75; % This overwrites the value stored in the variable final

>> average_exam_score = (midterm_1 + midterm_2 + midterm_3 + final) / 4;
```

Hint, you can hit the up arrow key on your keyboard to bring up your recent command history to select the previous line of code so you don't need to retype it! This is a valuable tool to remember.



The image shows a screenshot of the MATLAB Workspace window. It has a title bar 'Workspace' with a dropdown arrow. Below the title bar is a table with two columns: 'Name' and 'Value'. The table contains five rows of variables: 'average_exa...' with value '83.7500', 'final' with value '75', 'midterm_1' with value '86', 'midterm_2' with value '92', and 'midterm_3' with value '82'. Each row has a small yellow square icon to the left of the variable name.

Name	Value
average_exa...	83.7500
final	75
midterm_1	86
midterm_2	92
midterm_3	82

Figure 8.3: Updated workspace variables.

At this point, you can see that your `average_exam_score` variable has been updated to the value of `83.75`. If you were following along, the variables in your workspace should look like this (Figure 8.3 above).

Here is the problem. If you close MATLAB and reopen it all of those variables are lost (technically you *could* recover them from your command history but that is a hassle). What happens if we want to re-calculate your grade to see what would happen if you get a `95` on the final? You would have to retype all of those lines back in! Clearly, that is not very efficient. The power in MATLAB and most scripting programming languages is that you can make a script that can be changed and re-run with whatever variables you choose. The *script* is just a linear series of commands that MATLAB executes stored in a file with a `.m` extension.

Learning Goals

In this chapter, we will expand on the previous chapter to show you the real power of MATLAB. We will learn:

- How to write MATLAB programs (aka scripts) and how to save them.
- What an array is and why it is so important in MATLAB.
- How to create vectors with equal spacing
- How to create linearly spaced vectors
- How to create matrices
- What the transpose operator is and how to use it
- How to address vectors and matrices
- Two of the built-in functions MATLAB provides for handling and manipulating arrays

Script Files

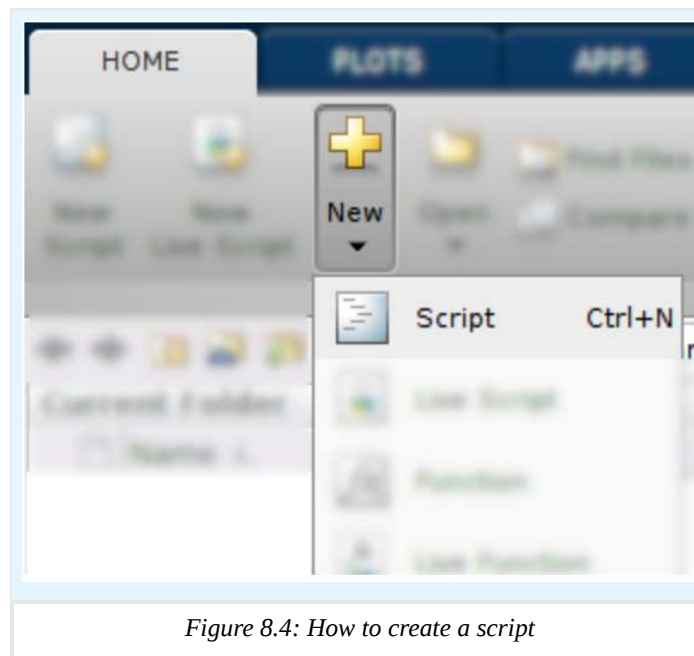


Figure 8.4: How to create a script

To work out our brain muscles, let's create a script that calculates the average exam grade as in the example presented above. As we have learned, a script file is just a list of MATLAB commands to be executed in a specific order. So in a way, we already have the meat for our first script file. To start a new script, click the "New" button on the "Home" toolbar and then select "Script" (Figure 8.4, everything that you don't need to worry about is blurred out).

Once you click "New Script", you should have a new window open titled "Editor" with a little pencil and paper icon in the top left corner. That is where we will write our script file. Go ahead and type in the commands we went over. When you are done, you should have something that looks like figure 8.5. Make sure that you typed all of the commands into the editor window correctly.

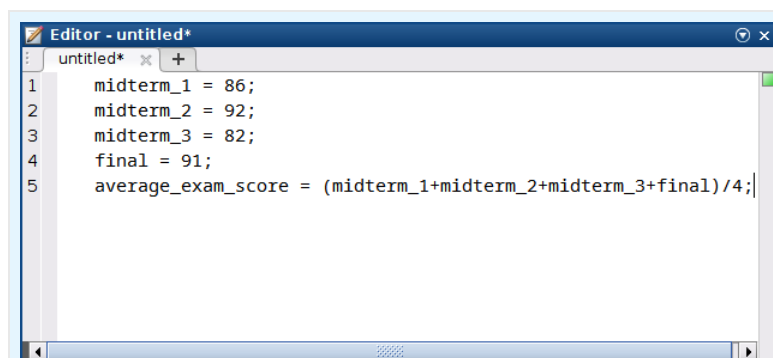


Figure 8.5: What your editor window should look like

Saving Scripts

Remember, the whole point of having scripts is so that we can save them and run them whenever we want and change the variables as we please. Notice that in Figure 8.5 that currently the editor is displaying "untitled*" in the window. This is mentioning to the user that the file currently does not have a name and the asterisks is indicating that the file is currently not saved. Hold your horses, don't go saving that file yet! We need to make sure we know **where** we are saving it.

In Question 8.1 above the current directory being displayed is `/home/sbechara/Documents/MATLAB` . That seems like an easy place to remember *for me*. **You need to decide where to store your file.** If I were you, I would create a folder in my documents folder called “MATLAB Book Examples” or something like that. Then navigate to that folder in MATLAB (did you forget how? No worries, you can click [here](#) to jump to the chapter of the book where that was discussed).

Now that you know where you want to save your file, click the little “Save” icon on the “Editor” tab. Notice that MATLAB automatically appends a `.m` to the file as the extension. I saved mine as `avgExam.m` . You should notice the file show up under the “Current Folder” portion of the workspace and notice that the asterisks are no longer listed next to the name (figure 8.6).

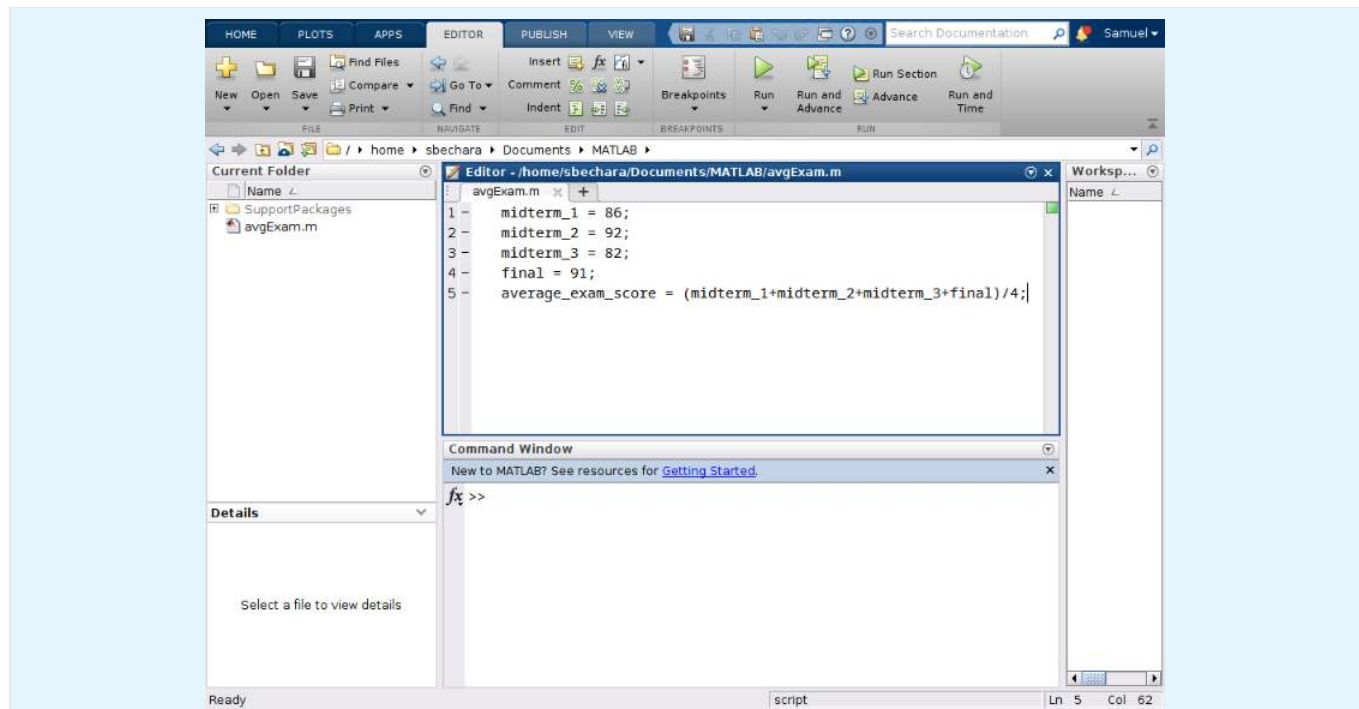


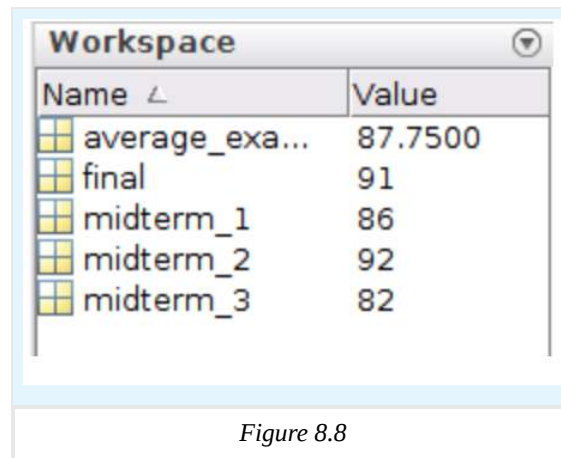
Figure 8.6: Notice that the current folder now has `avgExam.m` and that the asterisks are no longer next to the filename indicating that the file was successfully saved.

Running Your Script



Figure 8.7

Double-check that you have `avgExam.m` in your “Current Folder”. We are ready to run our first script! There are actually two ways to run the script. The first way is to click the green “Run” button on the “Editor” banner. Go ahead and click it now!



If you typed everything in correctly, the only thing that you should notice when the program is complete is that you now have several new variables in your workspace corresponding to your variables (specifically the `midterm_1`, `midterm_2`, `midterm_3`, `final`, and `average_exam_score` variables).

The preceding question is actually the *other* way that you can run your `.m` files. As long as the current working directory contains a `.m` file, you can simply type the name of the `.m` file into MATLAB (omitting the `.m` extension) and it will run the script!

Try It!

Remember, we need to work out your brain muscles to make them stronger and to ensure that you remember how to do this! **Do not skip these steps!**

- Try playing around with the different exam scores, saving, and re-running your script.
- Try navigating to a different folder. Then click the Run button. What does MATLAB do?
- Try clicking “Save As” and changing the file name. Try both methods you learned above to re-run the script.

Script Headers and Commenting

Commenting is a critical component of scripting because it allows us to understand what we did in the past so that we can work with scripts in the future. Do you remember the correct symbol for commenting in MATLAB? (If not you can click [here](#) to return to that chapter to review). Hint: `%`.

It is a good habit to create headers for your MATLAB scripts that contain useful information about your script. It may seem obvious what you are working on now, but open up this script a few weeks, months, or years, in the future and you will have no idea what it is for! That is why headers are so important. You can use comments because MATLAB ignores commented lines of code. I suggest that your script header contains the following information:

- The filename (redundant I know, but it helps in case something gets changed on accident)
- Your name

- The date the script was created
- The date the script was last modified
- A brief description of what the script does

In general, it is a good idea to comment on all of your script files so that you can follow your scripts in the future. Script headers plus comments play a huge role in helping future you understand what you were doing with a particular script. Remember, as soon as MATLAB sees % it ignores everything after the % on the same line. That means you can have in-line or whole-line comments. It also turns it into a nice green color for the user to have a visual indication that the line is commented out.

```
% Whole line comment
midterm_1 = 86;
midterm_2 = 92; % In line comment.
midterm_3 = 82;
```

Figure 8.9: Whole line vs in-line comments

This is where the *art* of commenting comes into play. How much commenting is too much or too little commenting? There really isn't a hard and fast rule with this. It depends. I suggest that you comment enough so that you understand what is happening in each chunk of code. What is a chunk of code? Again it depends. See figure 8.10 to see my script header and how I would comment on the `avgExam.m` script we have been working on.

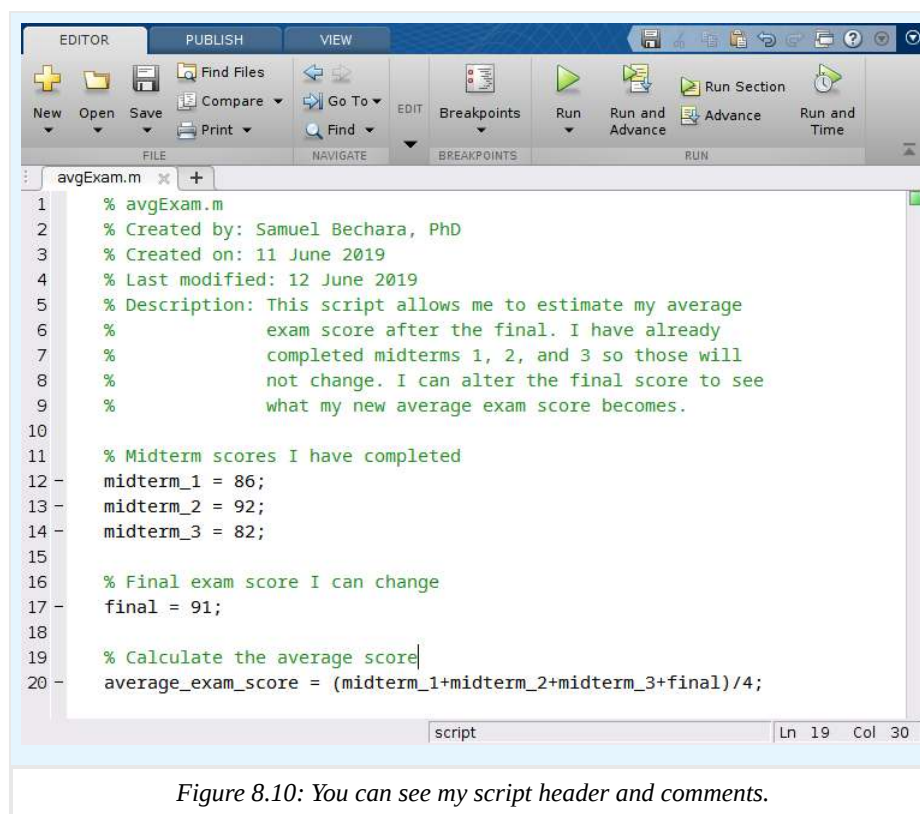


Figure 8.10: You can see my script header and comments.

Looking at figure 8.10 above, you can see how the header helps the user see what the script does and how it is intended to work. Keep in mind there are no right or wrong answers when it comes to commenting. Depending on where you work, your company

might have specific requirements about how much commenting and what headers should look like. The important thing, for now, is to make sure that from now on you are commenting on your script files and including some form of header so that you can understand what your scripts do in the future.

For now, you can save this script (we will use it in the next Chapter) and close it. For the rest of the chapter, we will just work out of the command window.

Arrays - The Fundamental Data Type of MATLAB

As we will see in future chapters, our script can be further improved using arrays, the fundamental data type in MATLAB. An array is a list of numbers arranged in rows and/or columns. In practice, **one-dimensional arrays are called vectors** and have numbers stored in either a row or a column. **Two-dimensional arrays are called matrices** and are collections of numbers stored in both rows and columns.

```
row_vector =  
  
    12    88    22  
  
column_vector =  
  
    12  
    88  
     2  
  
matrix =  
  
     8     1     6  
     3     5     7  
     4     9     2
```

Figure 8.11: Examples of different kinds of arrays.

Creating Vectors

To create a **row vector** in MATLAB, you use square brackets and input your data in the brackets using either spaces or commas:

```
row_vector = [vector elements go here] OR row_vector = [vector,elements,go,here]
```

Try typing the following line into the *command window*. Instead of spaces, you can put commas between the numbers if you prefer. I recommend trying it both ways for your brain workout. Do not copy and paste! typing it out will help with remembering:

```
1 final = [88 89 90 91 92 93 94 95 96 97 98 99]
```

runrun allrestartrestart & run all

When you hit enter, you should see MATLAB displays the row vector to you in the command window. Now we have a row vector that corresponds to a bunch of different scores for the final. This can be useful because it allows us to calculate a whole array of different averages based on the different exam scores! We will learn how to do that in the next chapter. For now, concentrate on learning the basics of arrays.

To create a **column vector** in MATLAB, you use square brackets just like before except you hit ENTER or use semi-colons between each data element:

```
column_vector = [vector;elements;go;here]
```

Try typing the following line into the command window. Like before, try hitting ENTER after each element instead for your brain workout):

```
1 final = [88;89;90;91;92;93;94;95;96;97;98;99]
```

run

run all

restart

restart & run all

Creating Vectors with Constant Spacing

When you were typing out `final = [88 89 90 91 92 93 94 95 96 97 98 99]` you may have found that tedious. It was! Whenever you find yourself doing something tedious there is a good chance that there is a better way to do it. MATLAB programming should never be tedious like that, it was designed for speed. Notice for the variable `final` as we defined it, the spacing is equal to 1 between each element.

To create vectors with constant spacing we can use the colon operator, `:` as follows:

```
constant_spaced_variable = [start:spacing:end]
```

In this case `start` is referring to the first term in the series, `spacing` is indicating how far apart you want your numbers to be, and `end` indicates the last number in the series.

Try It!

Try typing the following into the command window to see how it works:

```
1 final = [88:1:99]
```

run

run all

restart

restart & run all

I am sure you will agree that is a lot better! Brain workout time, try messing around with different numbers to see how MATLAB reacts. Try creating different equally spaced variables and name them different things. Spend a moment to see how they show up in

the workspace.

One last note when using the `:` operator. If you want to create an equally spaced array with spacing equal to 1, you can omit the `spacing` because MATLAB will default to 1. This can save you a little time.

For example,

```
1 final = [88:99]
```

run

run all

restart

restart & run all

will create an array from 88 to 99 spaced by 1 automatically.

Creating a Vector with Linear Spacing

Sometimes it is useful to create arrays that contain a *specific number* of values between two points. In these cases, it might be difficult to use the `:` operator. For example, we might need 12 equally spaced points between 2 and 4. You can do the math to figure out how to use the `:` operator but since that sounds tedious, you can bet there is a built-in MATLAB function to make our lives easier.

The `linspace()` function will do this for us. We can create these types of variables using `linspace()` as follows:

```
linear_spaced_variable = linspace(first,last,num)
```

This will create a variable called `linear_spaced_variable` starting at the number `first`, to the number `last`, with `num` number of elements. There are a few important things to note. Although we are creating an array, we do not use square brackets. That is because `linspace` is a function and functions are called using parenthesis. We will learn more about this in future chapters, but for now, just be careful about the parenthesis.

Let's try it with our example above by typing the following into the command window:

```
1 linear_boi = linspace(2,4,12)
```

run

run all

restart

restart & run all

```
linear_boi =
```

```
Columns 1 through 8:
```

```
2.0000    2.1818    2.3636    2.5455    2.7273    2.9091    3.0909    3.2727
```

```
Columns 9 through 12:
```

```
3.4545    3.6364    3.8182    4.0000
```

Notice how the array is equally spaced and that there are exactly 12 elements in the array. Brain workout time. Play around with `linspace` on your own. Try different combinations of numbers and variable names. Notice how they appear in the workspace. **DO NOT SKIP YOUR WORKOUT!***

```
1 % Here is a code box to try using the linspace function
2
3 test_array = linspace()
```

run

run all

restart

restart & run all

Creating Matrices

Recall that a matrix has numbers in both rows and columns. They are used extensively in science and engineering and are useful for storing numbers in table-like formats or for other linear algebra techniques that you will learn in future classes.

The terminology we will use in this class is to describe matrices by the number of rows and columns they have. Furthermore, you can have either square or non-square matrices depending on the number of rows and columns. Consider the following two matrices:

```
square =
```

```
2  4  5
2  1  2
6  3  5
```

```
not_square =
```

```
2  3  1
1  1  3
7  9  2
2  1  1
```

Figure 8.12: Example matrices

In figure 8.12 above, we can see that the matrix `square` has 3 rows and 3 columns. To make it easy to write, it is common to say that this is a 3x3 (pronounced “three by three”) matrix. Square matrices have the same number of rows and columns. It follows that since `not_square` has 4 rows and 3 columns it is not square. We would say this is a 4x3 matrix (pronounced “four by three”).

Creating matrices in MATLAB is as easy as combining what we have learned about creating row and column matrices above. To create a matrix you type:

```
matrix_name = [1st row elements; 2nd row elements; ... ; Last row elements]
```

Try It!

Let's try making our own matrix by typing the following into the command window:

```
1 neo = [1 2 3; 4 5 6; 7 8 9]
```

run

run all

restart

restart & run all

Notice that this creates a 3x3 matrix called `neo`. Brain workout time. He had a `64.99` which rounds to `65` according to my course policy. I also state that a `66`

The only difference between matrices and vectors is that *all the rows must have the exact same number of elements*. For example, you can't have row 2 have 3 elements, while row 3 has 4 elements.

Creating Arrays of all 0's or all 1's

Sometimes it is useful to create vectors or matrices with all zeros or all ones in their spots. You *could create* a 4x4 matrix with all ones by typing this:

```
1 tedious_ones = [1 1 1 1; 1 1 1 1; 1 1 1 1; 1 1 1 1]
```

run

run all

restart

restart & run all

You can imagine that creating a 4x4 matrix with all 0's would be equally as annoying. But as we have learned, if something sounds tedious, there is a better way to do it. MATLAB has built-in functions that will do this automatically for us.

To create an array of all 1's you use the MATLAB function `ones()` as follows:

```
all_ones = ones(rows,columns)
```

Similarly, to create an array of all 0's you use the MATLAB function (you guessed it) `zeros()` as follows:

```
all_zeros = zeros(rows,columns)
```

In both cases, you only specify the dimensions of the output array. For example, if you want a variable named `skinny_boi` to be a 5-element column vector with all ones you would type in:

```
1 skinny_boi = ones(5,1)
```

run

run all

restart

restart & run all

As always, do your brain workout. Play around with both the ones and zeros commands to make sure that you can create row vectors, column vectors, and matrices using the commands. As usual, store them in different variable names and notice how they

appear in the workspace.

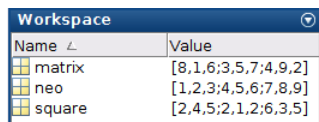
The Transpose Operator

Remember, that if I want a row vector variable called `rex` containing the elements 2,4,6,8,10 I can simply type in `>>rex = [2:2:10]`. But what happens if I want a column vector that contains those elements? Tedious alert! That means there is a shortcut. In this case, we are talking about the transpose operator `'`.

The transpose operator will switch the row to columns of any type of an array. That means if you apply the transpose operator to a column vector, it will turn into a row vector. If you apply the transpose operator to a matrix, it will switch the rows with the columns.

If you ran the `>>rex = [2:2:10]` line of code above in the MATLAB command window you can create a new column vector named `tex` by typing in `>> tex = rex'`.

Given the image of the user's workspace, consider the command `trinity = neo'`.



Name	Value
matrix	[8,1,6;3,5,7;4,9,2]
neo	[1,2,3;4,5,6;7,8,9]
square	[2,4,5;2,1,2;6,3,5]

Array Addressing

You might already be able to see why arrays are powerful tools to utilize in MATLAB. For example, we could have arrays that correspond to the position of a moving object with respect to time. Let's pretend that we have the following position and time data. Go ahead and define the following variables in your workspace by typing them into the command window.

```
1 time = [0:5:20] % time in seconds
2
3 position = [0, 4, 8, 12, 30] % position in meters
```

run

run all

restart

restart & run all

You might recall from physics that we can calculate the average velocity between two points in time using this data. We can see that the object starts at time 0 at position 0. It then moves in the positive direction 4 units in 5 time units. We could calculate the average velocity of the object for any time period by:

$$avgVelocity = \frac{x_2 - x_1}{t_2 - t_1} \quad (4.2.2)$$

where x_2 and x_1 correspond to positions and t_2 and t_1 correspond to times.

Since we have already created the arrays `time` and `position` in MATLAB we can use array addressing to pull out individual numbers within the array.

Vector Addressing

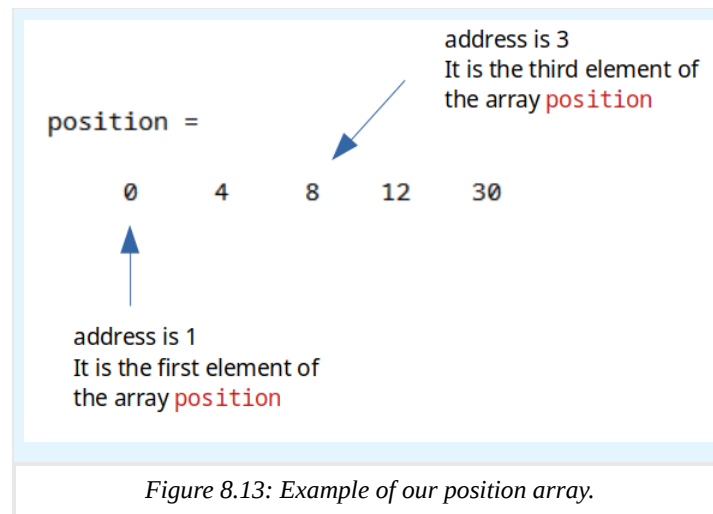


Figure 8.13: Example of our position array.

In this example, our array `position` has 5 elements in it corresponding to the numbers 0, 4, 8, 12, and 30. In this example, the number 8 has an address of 3. It is the third element in the array `position`. To access this number, we can call the array `position` and specify that we want the third element out of it.

Try typing this into the command window:

```
1 time = [0:5:20]; % time in seconds
2
3 position = [0, 4, 8, 12, 30]; % position in meters
4
5 position(3)
```

run

run all

restart

restart & run all

You should notice that MATLAB stores the number 8 into the `ans` variable. We have accessed the number 8 from the `position` array! It may not be obvious now (some of you may be asking, why not just type in the number 8?) but believe me when I say this is a fundamental skill to have in MATLAB and is critical to your future success!

See if you can guess what the output of the following command will be, before typing it in:

```
1 time = [0:5:20]; % time in seconds
2
3 position = [0, 4, 8, 12, 30]; % position in meters
4
5 anyone_home = position(4) + 4
```

run

run all

restart

restart & run all

What happens when you type that in? You create a new variable called `anyone_home` and it stores the number 16 in it. Why 16? Because `position(4)` accesses the 4th element in the array `position`!

Let's go back to our position and time arrays and calculate the average velocity of the object between times 15 and 20.

As you can see, addressing values inside of vectors is as easy as typing in the vector name, followed by parenthesis, and then typing in the number that corresponds to its position!

Try It!

Brain workout time. Try creating different size row and column vectors with random numbers and use array addressing to make sure that you understand how to pull out the correct values. Array addressing on row and column vectors works the exact same way. The addresses start at 1 and go from there.

```
1 % Use this code box to construct your practice arrays
```

run

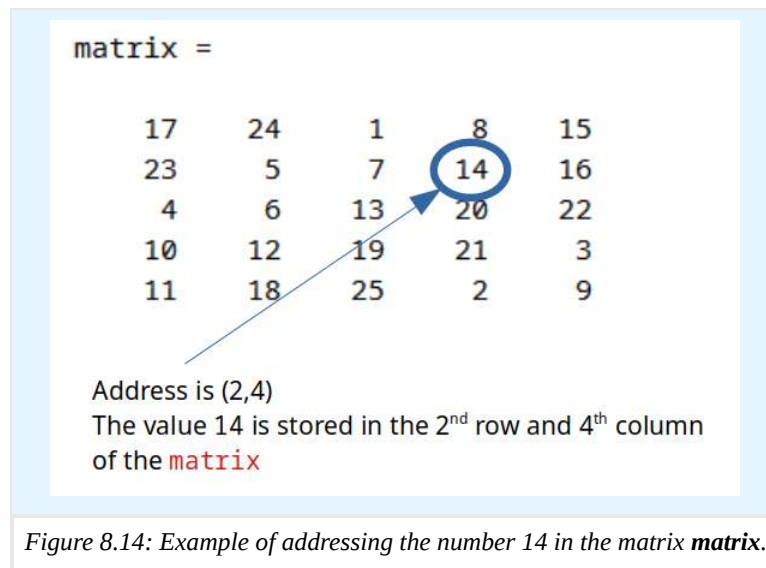
run all

restart

restart & run all

Matrix Addressing

Matrix addressing works the same way as vector addressing, the only difference is that with matrices we need to specify *both the row and the column* of the value of interest. Take a look at figure 8.14 for an example.



Go ahead and enter the matrix in figure 8.14 into your workspace. It is good practice to remember how to enter all of the values. When you are done double-check that your matrix is the same as mine.

Now that you have done that, let's try accessing the value 18. We can see that it is in the 5th row and the 2nd column of the matrix. That means, to get the number 18 we can type in:

```
>> matrix(5,2)
```

And now you know array addressing for both matrices and vectors! The main trick to remember is that when we are talking about array addressing (and honestly pretty much everything dealing with a matrix in MATLAB) it always goes **ROWS then COLUMNS**.

Try It!

You know the drill. Brain workout time. I suggest that you make a random 4x5 matrix on a sheet of paper. Make sure that you can successfully type that matrix into MATLAB. Next, use array addressing to make sure that you can pull out numbers successfully. The nice thing about not using a square matrix is, that it will show you if you really understand the row/column law of MATLAB.

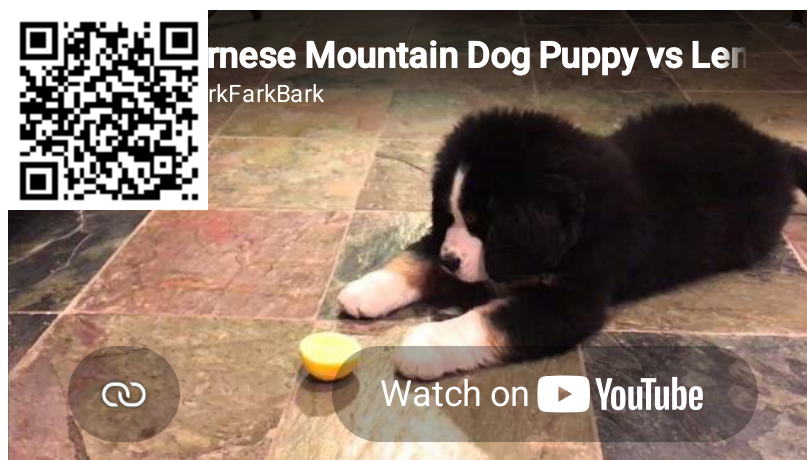
Quick Recap on Arrays

Up to this point, you should be able to do the following with arrays:

- Create row vectors.
- Create column vectors.
- Create equally spaced arrays using the `:` operator.
- Create linearly spaced arrays using the `linspace()` function.
- Create matrices.
- Create arrays using all 1's and all 0's using the `ones()` and `zeros()` functions.
- How to use the transpose operator `'`.
- How to address arrays and access the values they contain.

Whew! That is a ton. Good work!

We have one more topic to go over but you have been working out your brain hard and your brain muscles are probably sore. You could use a one-minute break. Here is a video of a Bernese mountain puppy barking at a lemon.



Video 1: Totally unrelated but you need a break now and then!

OK, that was cute. Brain muscles relaxed? Ready for one last set of our brain workouts? Good, there is one last topic for this chapter...

Advanced Array Addressing with the `:` Operator

You can couple what we have learned about the `:` operator with addressing for arrays to do some really cool stuff. **The key thing to remember is that the `:` operator defaults to a spacing of 1!** That means **you can use the `:` operator to address a range of elements in a vector or a matrix.**

Using `:` with vectors

As mentioned above, we can use the `:` operator to address a range of elements from a vector. The most general use case is as follows:

```
range_from_vector = vector_name(m:n)
```

This would take out the m^{th} through the n^{th} elements of vector `vector_name` and store them in a new vector called `range_from_vector`. It's that easy!

For this example, let's add a random vector `v_man` to our workspace as follows:

```
1 v_man = [22 61 45 51 68 98 11 90 17]
```

run

run all

restart

restart & run all

We can use the `:` operator to address different ranges of elements from the vector `v_man`. Let's say for this application we want to create a new variable called `two_to_five` and we want to store the 2nd through the 5th elements of vector `v_man`.

```
1 v_man = [22 61 45 51 68 98 11 90 17];  
2
```

```
3 two_to_five = v_man(2:5)
```

runrun allrestartrestart & run all

That will accomplish our goal!

Before you try that on your computer, do you know what those elements' values should be? Take a second to write them down before trying to make sure you understand array addressing!

Using `:` with Matrices

Because matrices have two dimensions, there are many more ways that we can use the `:` operator to address ranges of elements within a matrix. However, the logic is the exact same as with vector addressing. Again, **the key is to remember the row/column law of MATLAB.**

Hopefully, you still have some matrices stored in your workspace. Before you continue, see if you can guess how to use the `:` operator on matrices to pull out a range of values. Can you figure out at least a couple of ways to make it work?

If you couldn't figure any out, *that is OK!* The fact that you tried is all that matters.

Here are some of the ways that you can address a range of elements within a matrix. For this example, let's consider the general case of a matrix `A`. It can have any number of rows and columns.

- Refer to the elements in *all rows* of *column n* of matrix `A`.
`A(:, n)`
- Refer to the elements in *all columns* of *row m* of matrix `A`.
`A(m, :)`
- Refer to the elements in *all the rows* between *columns m and n* of the matrix `A`.
`A(:, m:n)`

Refer to the elements in *all the columns* between *rows m and n* of the matrix `A`.

`A(m:n, :)`

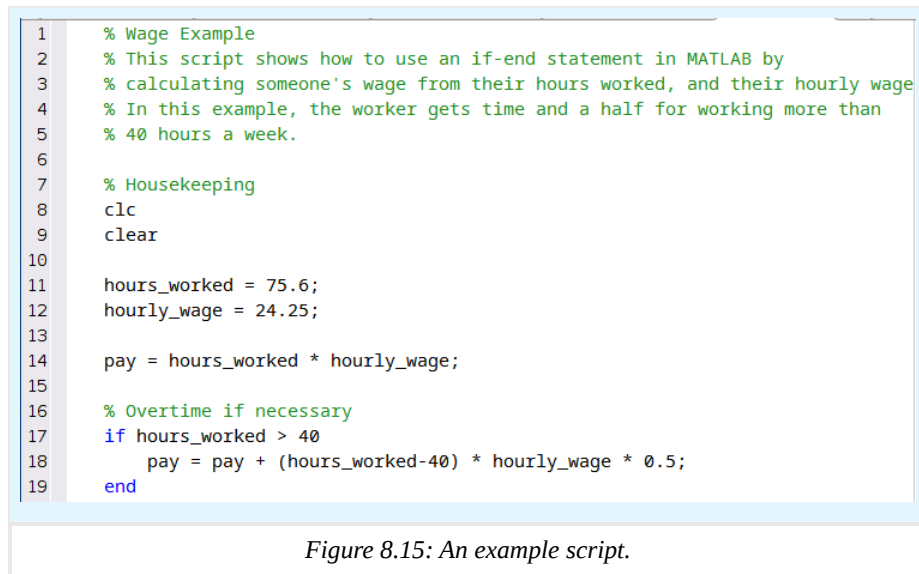
Refer to the elements in *rows m through n* and *columns y through z* of the matrix `A`.

`A(m:n, y:z)`

For your brain workout for this section, try every one of the bullet points above on one of the matrices we have previously defined to make sure you understand how this works!

Housekeeping

The last thing I would like to mention about scripts is that it is a good idea to include what I call “housekeeping” lines of code at the **top** of your script. Most of my scripts start with `clear` and `clc`. For example, look at a script that we will work on in Chapter 13 below in figure 8.15.



Don’t worry about what this script does (yet). Instead, notice how I started the script with a `clc` and then a `clear` command. This is good practice to ensure that your scripts will work on a *fresh start* of MATLAB and do not rely on a variable that is currently in the workspace. You are not required to start your scripts this way, but it is good practice.

This page titled [4.2: MATLAB Scripting and Arrays](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 08: MATLAB Scripting and Arrays](#) is licensed [CC BY-NC 4.0](#).

4.2.1: MATLAB Decisions



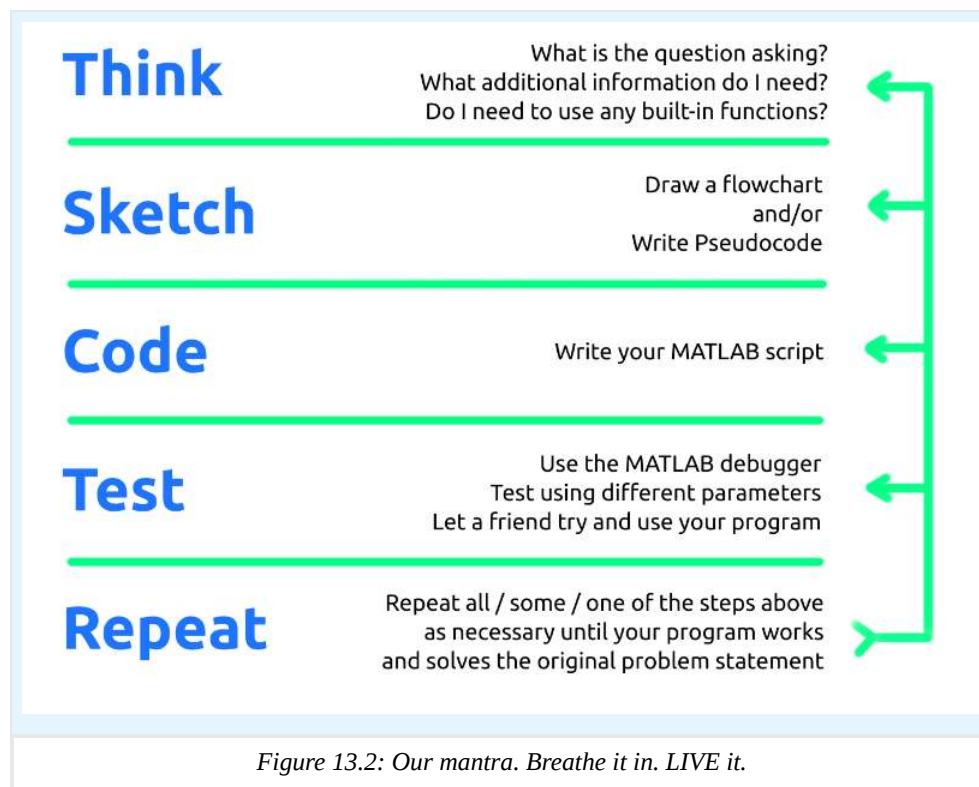
Figure 13.1: If we want our computers to be “smart” we need to have a way for them to make decisions.

So far, the programs we have been writing are great at performing calculations but they are limited and not like software that you are used to using. We are used to software making “decisions” based on information. For example, when it is going to rain, Google Assistant (or Siri, or Alexa, or whatever) can send you an alert reminding you to bring an umbrella. When it is going to be sunny, that same assistant might suggest wearing sunscreen. The software underpinning those technologies has the capability to make decisions about what it tells you depending on the weather. I am certain that you can think of several other instances of computer programs making decisions.

Briefly, share with your classmates a unique way that a computer program makes a decision in some software that you regularly use. You can pick any computer program you like! For example, one of my favorite video games growing up was Mario Kart 64. My sister was WAY better than me at racing but the power-ups that you can collect change depending on what place you are in. Since I was usually in last place, and my sister was usually towards the front, the program made decisions to give me much more powerful items that allowed the race to be competitive! What is your story about computers making decisions?

The reason the scripts we have written so far seem so limited is that we haven’t learned a way to control the **flow** of programs or have them make any decisions. Controlling the flow and allowing our programs to make decisions is what we will learn in this chapter.

Our mantra and guiding process for writing complex programs will be: **think, sketch, code, test, repeat**. We will use what we learned about thinking algorithmically to make sure that we **think** before we do anything. We will **sketch** to ensure that we come up with a robust plan. After coming up with a sketch, we will write our **code**. Then we will test the code using as many absurd and wide-ranging possibilities as we can think of. Finally, we will **repeat** the entire (or part of) the process until our code works the way we want it to!



Learning Goals

In this chapter, we will learn: how to combine what we learned about thinking algorithmically with MATLAB syntax to make complex computer programs. That means learning:

- How to combine conditional statements with MATLAB `if` structures.
- How to use flowcharts to help understand our code.
- The specific syntax necessary to code `if` and the various derivatives of an `if` statement.

Quick Note Before We Start

In my opinion, it is critical to follow along with this chapter, even more so than in previous chapters. Even if you have experience in other programming languages, the way that MATLAB handles loops and decisions is different. Make sure that you start MATLAB, create a new script, and follow along! It will help in two ways. First, you will get the experience from working on it today. Furthermore, by saving your script you can create good notes for future reference.

Conditional Statements

Now we are going to combine everything we have learned about MATLAB to allow our programs to make decisions. The trick is to combine what we learned about the relational and logical operators and thinking algorithmically to create what are called conditional statements.

A **conditional statement** is an **if-then** statement (there are a few other flavors that we will learn about shortly, but this is the most simple one). We already saw examples of these types of statements in the thinking algorithmically chapter, but here are a few examples of conditional statements to jog your memory:

- **If** I skip my brain workouts, **then** I will not understand the material.

- If I finish my homework on time, then I will receive a high grade on homework.
- If it is raining outside, then I will wear a jacket.

MATLAB if-end Statements

The most basic form of a conditional statement is the If-End statement. Generically in MATLAB, an If-End statement looks like this:

```
>> if < conditional expression is true >  
>> < do this stuff >  
>> end
```

Let's break this down:

- The MATLAB conditional statement *must* start with the keyword `if`
- The `if` is followed by a conditional expression. Recall that the `< >` indicate that you need to replace this with actual MATLAB code. A conditional expression is simply an expression with at least one relational operator and any number of logical operators. For example: `x > 12` is a conditional statement. So is `angle > 25 & angle < 100`.
- Next, we have `< do this stuff >`. This is simply the MATLAB code that we want to execute *only if the conditional expression evaluates to true*. MATLAB will automatically tab the text within the decision structure over to make it easy to read.
- Finally, the decision statement *must* be completed with the keyword `end`.

The generic flowchart for this can be found in figure 13.3 below:

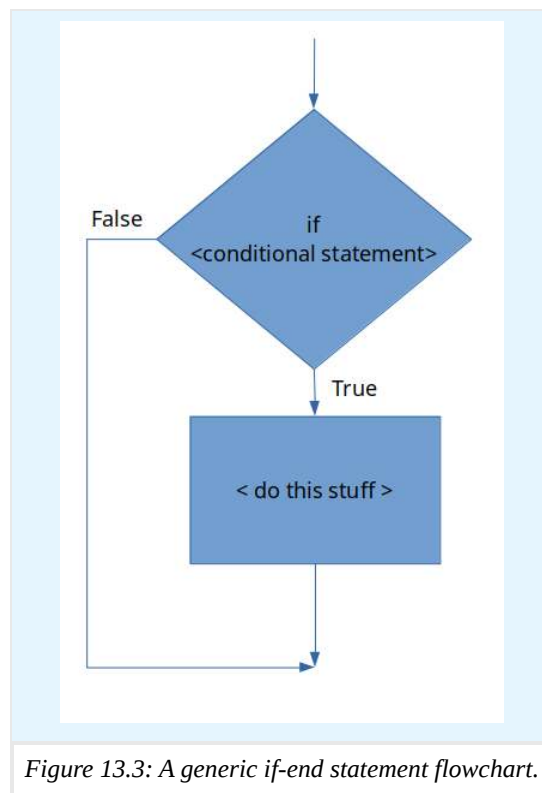


Figure 13.3: A generic if-end statement flowchart.

Example - Hourly Worker Calculation

To see this in action let's create a short script that calculates the weekly wage of an hourly worker. This program can be used to calculate the `pay` based on the number of hours worked and the worker's hourly wage using two variables: `hours_worked` and `hourly_wage`. If the worker works over 40 hours a week, they are compensated at 1.5x their normal hourly rate for *only the hours worked over 40*. For example, if a worker worked 42 hours for \$10 an hour, they would get paid \$10 per hour for the 40 hours and \$15 per hour for the extra 2 hours over 40.

This is a perfect example of using our mantra: **think, sketch, code, test, repeat!** Do not skip your brain workout for these sections. For each of the prompts below, I am expecting you to stop and actually work on this particular problem before moving on.

Stop & Think

Think - during the **think** phase you need to ask yourself and then answer a bunch of questions. I can't think for you but I can give you an example of my thinking process when solving the problem. Keep in mind that you might have other questions that need answering! That is OK! This is what I came up with as I was thinking.

- Do you understand what the problem is asking?
- What math is necessary to complete this problem?
- Can you calculate expected answers based on a few different variables? That way you will have numbers to compare your program against to make sure it works. Example: if `hours_worked = 42` and `hourly_wage = 10`, we would expect `pay = 430`.
- How can you have the program calculate the correct value *algorithmically*? Remember, that means you need to write down the process into steps. Start with calculating the `pay` in the situation for under 40 hours a week. Then move on to what would happen at 41 hours a week...
- What conditional statement, if any, is needed?

Sketch - during the **sketch** phase, you need to draw a flowchart or write out a pseudocode for your algorithm that will solve this problem. The **sketch** phase will often require several **repeats** of the **think** stage. As you are sketching more questions may pop into your head about how to best solve this problem. Before scrolling down below, take a minute and try to sketch out a flowchart or write out pseudocode for this particular problem. Don't skip out on giving it a try!



Figure 13.4: Every time someone scrolls and looks at my flowchart before creating their own, this puppy doesn't get a snuggle! Don't be a monster! I was going to say the puppy gets hit but that is too dark. Look how cute he is!

The flowchart that I created for this particular problem is located below in figure 13.5.

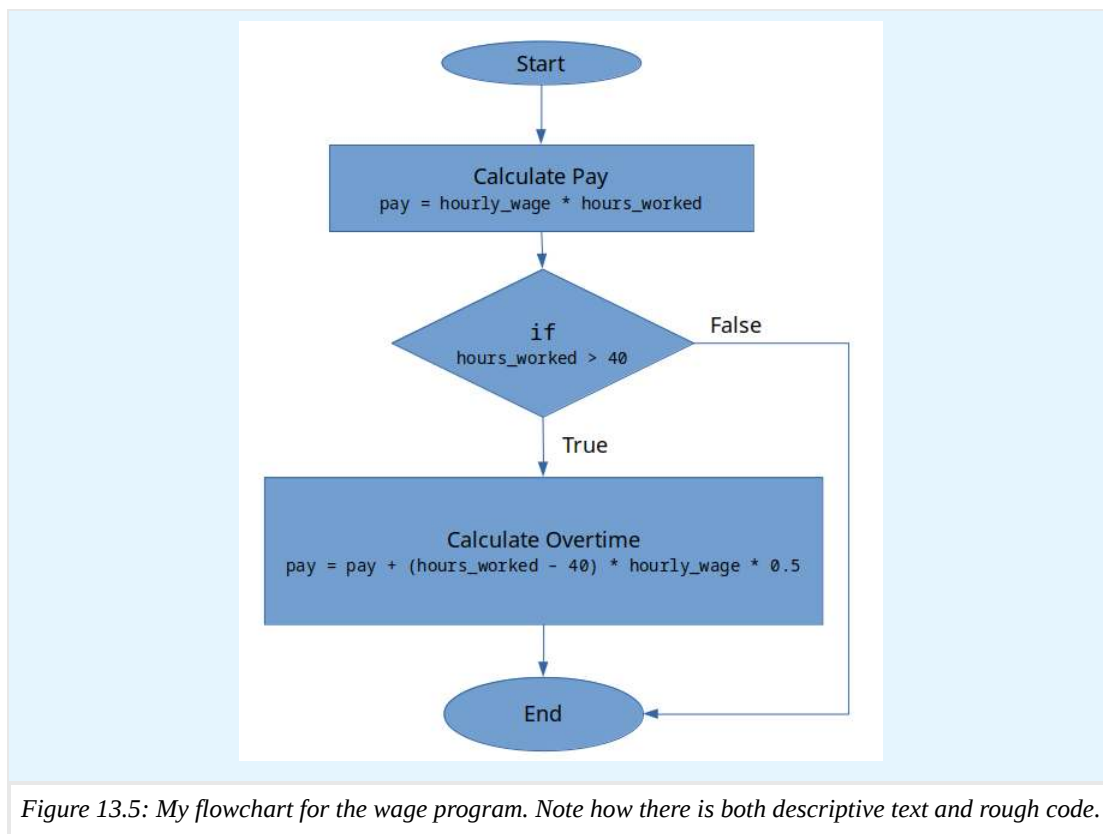


Figure 13.5: My flowchart for the wage program. Note how there is both descriptive text and rough code.

Take a look at the flowchart. Hopefully, it looks very similar to yours. Ask yourself the following questions before moving on.

- Do you understand how the flowchart works?
- Can you follow the logic?
- Why is the calculation beneath the “Calculate Overtime” step multiplying by 0.5 and not 1.5?

Do not move on to the next section until you understand the flowchart above in figure 13.5.

Code - Now comes the fun part, translating our **sketch** into **code**! I highly encourage you to try this on your own before you look at my solution below! It is OK if you get stuck and struggle! That is what learning is about. Take a moment to review the generic version of the `if-end` statement (in Chapter 16) so you have a good idea of the MATLAB syntax necessary to complete the problem.



*Figure 13.6: The lemur of judgment is majestically judging those who scroll down before trying to **code** on their own. Don't let the lemur down!*

Did you try it on your own first? Good! Does your script work? If yes, good work! If not, it is OK! Struggling and practicing *IS* learning!

```

1 %% Chapter 13 Work
2 % Created on: 12 June 2019
3 % By; Samuel Bechara, PhD
4
5 %This script contains the work that needs to be completed to follow along with handsoneengineering text
6
7 % Wage Example
8 % This example shows how to use an if-end statement in MATLAB by calculating someone's wage from their
9 % In this example, the worker gets time and a half for working more than 40 hours a week
10
11 hours_worked = 42;
12 hourly_wage = 10;
13
14 pay = hours_worked * hourly_wage;
15
16 % Overtime if necessary
17 if hours_worked > 40
18     pay = pay + (hours_worked - 40) * hourly_wage * 0.5;
19 end
20
21 pay

```

run

run all

restart

restart & run all

```
pay = 430
```

Figure 13.7: Example of my script file that calculates the pay. Notice I used sections (%%) to keep things organized and I have nice headers and comments explaining what each section does.

When you run the script with these example numbers, `hours_worked = 42` and `hourly_wage=10` , you can see that the value of the `pay = 430` !

Test - the testing phase might be the most fun. Just spend some time with the script to double-check that it works. Try substituting different values for variables and verifying by hand that the calculation works. This is an important step and becomes even more important the more complicated your scripts become! Just play around with it and make sure it works the way that you expect.

```
1 % Fix the code and use it to solve Question 13.1
2 % This script contains two errors that prevents the code from running properly
3 hours_worked = 75.6;
4 hourly_wage = 24.25;
5
6 pay = hours_worked * hourly_wage;
7
8 % Overtime if necessary
9 if hours_worked < 40
10     pay = pay - (hours_worked - 40) * hourly_wage * 0.5;
11 end
12
13 pay
```

run

run all

restart

restart & run all

Hopefully, by this point, you understand how the if-end statements are structured and how they are written in MATLAB. The reality is that there has been very little new material in this chapter. We have simply learned how to apply previous concepts to new problems. There are a few other decision structures MATLAB has that we should investigate.

The rest of the MATLAB conditional statements will be explained below and some example code will be provided but the **think, sketch, code, test, repeat** process will not be replicated in this much detail. The good thing is that the other MATLAB conditional statements are very similar and follow almost the exact same syntax.

MATLAB if-else-end Statements

There are instances when writing computer programs where it is beneficial to make decisions that will run one set of instructions if the conditional statement is true, and it will run a different set of instructions if the conditional statement is false. This is called an

`if-else-end` statement.

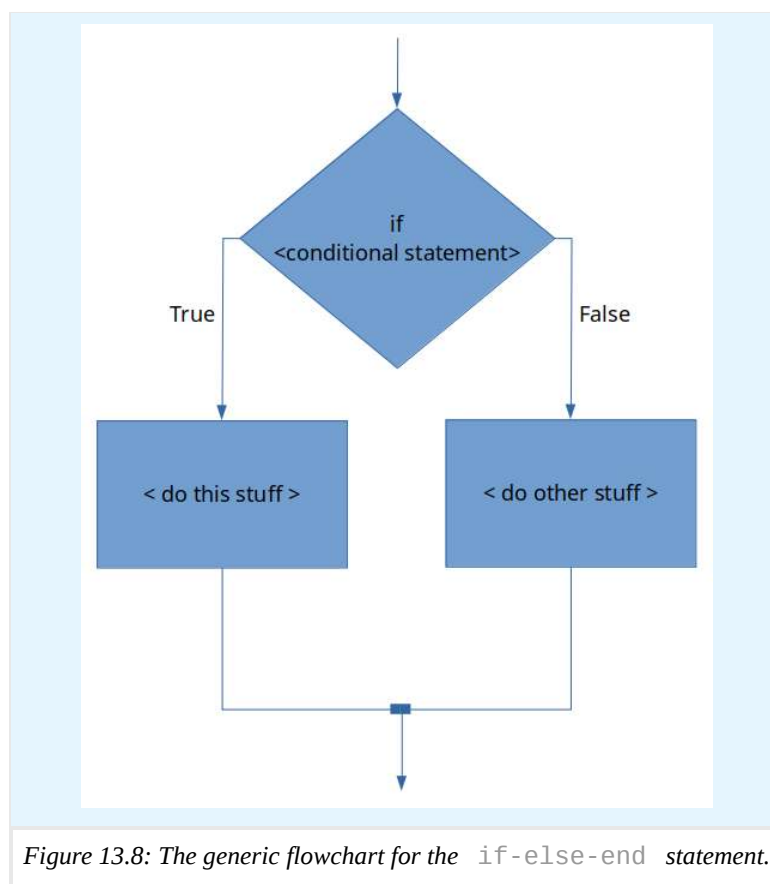
Generically, an `if-else-end` statement looks like this:

```
>> if < conditional expression is true >
>> < do this stuff >
>> else
>> < do other stuff >
>> end
```

Breaking this down:

- Notice that it is almost identical to the `if-end` statement with the exception that there is an `else` keyword.
- In this case, we have created two different branches. The `< do this stuff >` runs *only if the conditional statement is true*. Conversely, the `< do other stuff >` will only run if the conditional statement is false. In this case, it is not possible to run both `< do this stuff >` and `< do other stuff >`.

The generic flowchart for an `if-else-end` statement can be seen in figure 13.8 below.



You can see that the flowchart for an `if-else-end` statement is almost identical to an `if-end` statement, with the exception that an `if-else-end` statement can have two independent code blocks. Only one of the code blocks will be run depending on if the conditional statement evaluates as true or false.

Another thing to remember is that although our conditional statement was fairly simple in our wage example, conditional statements can be complex! For example, try this problem.

MATLAB if-elseif-else-end Statements

Previously, we just saw how we can use an if-else-end statement to essentially create two blocks of code that can be run independently depending on the outcome of the conditional statement. There is another MATLAB conditional structure that is even more flexible and actually allows you to have however many independent code blocks you would like. The if-elseif-else-end statement allows this.

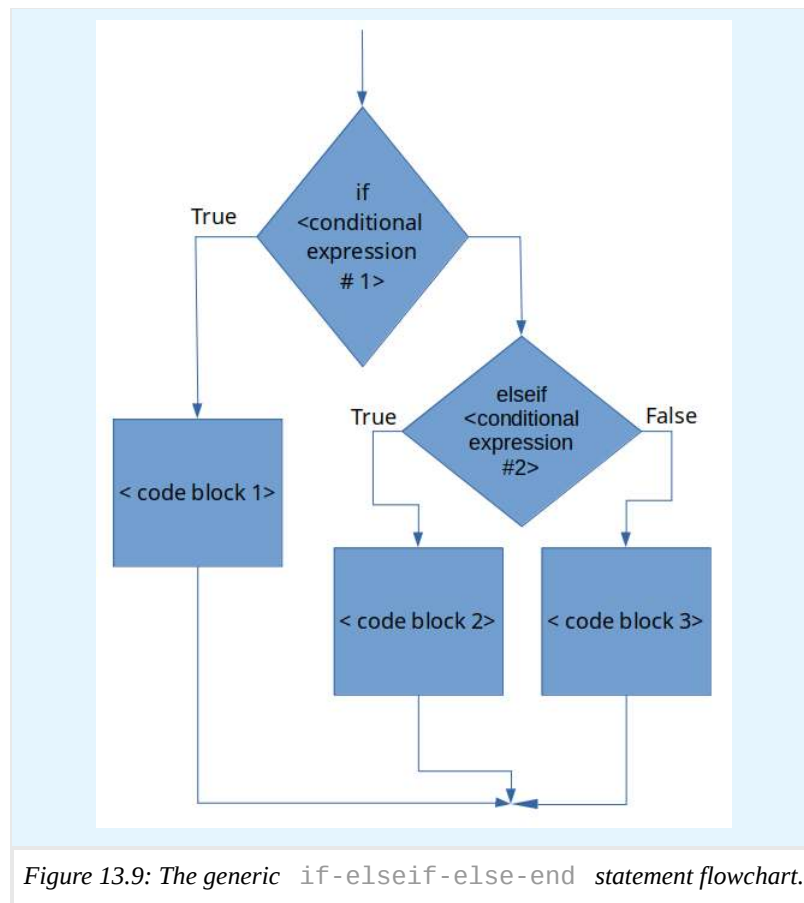
Generically, the if-elseif-else-end statement looks like this:

```
>> if < conditional expression #1 is true >
>> < code block 1>
>> elseif < conditional expression #2 is true >
>> < code block 2 >
>> else * technically the else here is optional
>> < code block 3 >
>> end
```

As you can see, this is very similar to the statements we have been covering thus far. However, there are a couple of things to note:

- Technically, the `else` is optional in an `if-elseif-else-end` statement. If you look at the flowchart in figure 13.9 below, omitting the final `else` would be equivalent to chopping off the `< code block 3 >` chunk of code.
- In this generic example there is only one `elseif` statement with a corresponding conditional expression. However, MATLAB lets you have *however many elseif statements that you need*.
- Note that because of the logic of the statement, you only need 2 conditions to run 3 separate blocks of code. In general, when designing an `if-elseif-else-end` statement that has (n) different conditions, you need (n-1) conditional statements.

The generic if-elseif-else-end flowchart is shown below in figure 13.9. Again, keep in mind it can have *any number* of `elseif < conditional expression >` lines!



Student Grades - if-elseif-else-end Example

I think that it is important to clarify the statement “when you have (n) different conditions, you need ($n-1$) conditional statements”. To do so, let’s consider an example. Let’s say that we have a variable `grade` that contains the final numeric grade for a person in a class. Furthermore, let’s say that this hypothetical class does not have + or - grades, instead, the final letter grade is calculated as follows:

Grade Range	Letter Grade
<code>grade > 90</code>	A
<code>80 < grade < 90</code>	B
<code>80 < grade < 70</code>	C
<code>grade < 70</code>	F

Table 13.1: Student grading scheme example.

The problem is to create a MATLAB script that can take the numeric grade as a variable `grade`, determine the letter grade associated with it, and store that in a variable called `letter_grade`.

This may seem contrived because, in this particular example, we will only have 1 grade but in the next chapter, we will learn techniques that will allow us to analyze any number of grades in the same fashion. What we are working on now will be the bones of that future algorithm!

Stop & Think

Think and Sketch - we can see that we have 4 different categories of grades. That means that ($n=4$) (the grade can be an A, B, C, or F). Before you continue, can you create the flowchart necessary to complete this script? Take your time. I am hoping that by thinking and sketching it will be obvious why you only need ($n-1 = 3$) conditional statements.

If you are feeling like a tough workout today, go ahead and try to program this script on your own. Go you! If your brain is too tired from that it is OK, but don't skip the flowchart part.

I won't threaten you with puppies or lemurs anymore. I will trust that you care about your education and want to learn this material. Handshake?

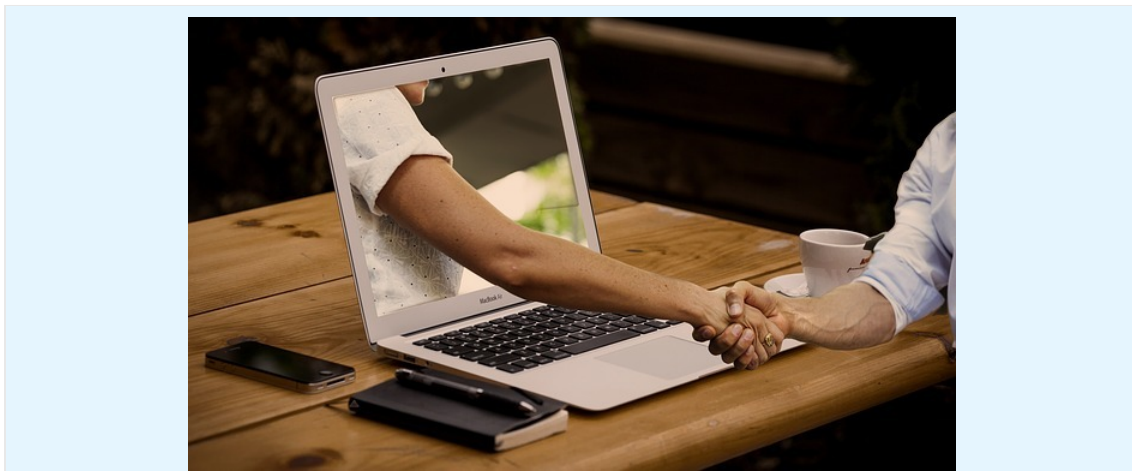
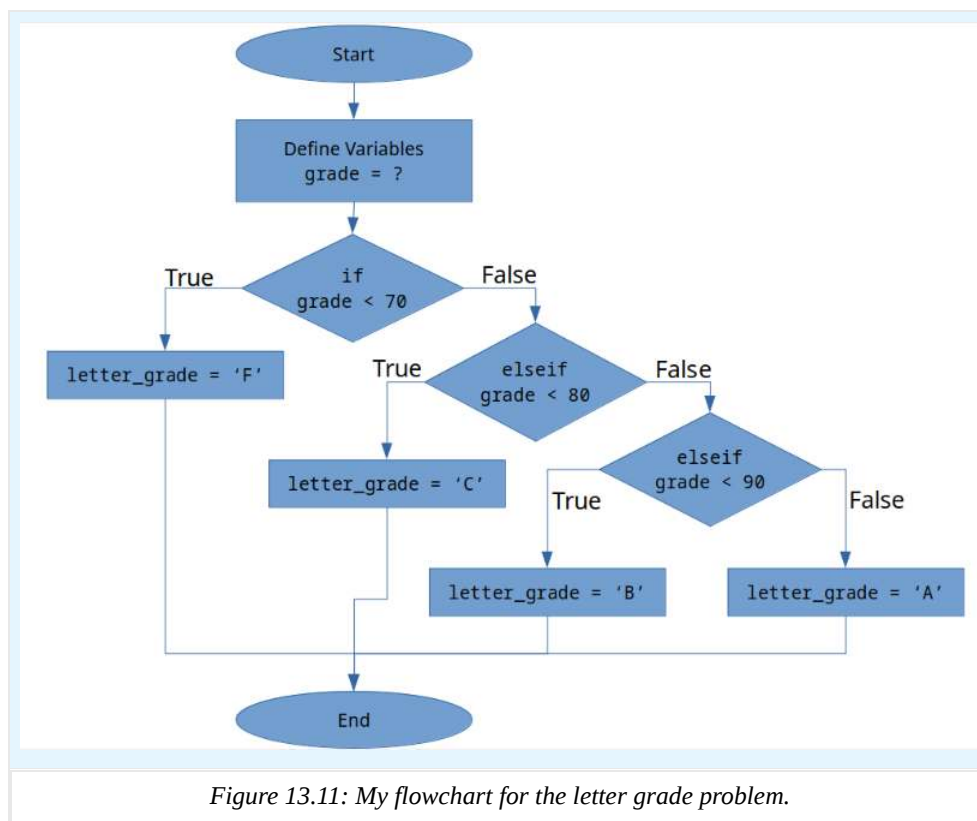


Figure 13.10: Our handshake agreement that you are working on the material. I love stock images so much.

Here is my flowchart for the problem.



Take a moment to compare our flowcharts.

- Did you start from high to low? That is a common way of doing things and totally fine. Maybe I am just weird? It doesn't matter.
- Do you notice how my flow chart only has three conditional statements? THAT is what I meant by (n) conditions requires (n-1) conditional statements. Here is the logic:
 - `if` you scored less than 70, you get an F
 - `elseif` you scored less than 80, you get a C
 - `elseif` you scored less than 90, you get a B
 - `else` , you get an A. (You do not need a conditional statement, because if you didn't get an F, C, or B, the only option left is an A!)

Try It!

Now is your chance to try and do the code, test, and repeat part of our process on your own to finish the problem! Even if you can't get it working perfectly, take some time and give it your best shot! Make sure you have a pen and paper handy and jot down notes, ideas, and snippets of code as you are working. You got this!

```
1 % Here is a code box for you to try this problem
```

run

run all

restart

restart & run all

My script solution follows in figure 13.2.

```
1 % Grading Script
2 % This script takes a 'grade' and assigns a letter grade based on a rubric presented in chapter 13
3
4 grade = 88; % can change as necessary
5
6 if grade < 70
7     letter_grade = 'F';
8 elseif grade < 80
9     letter_grade = 'C';
10 elseif grade < 90
11     letter_grade = 'B';
12 else
13     letter_grade = 'A';
14 end
```

run

run all

restart

restart & run all

Figure 13.12: My script solution.

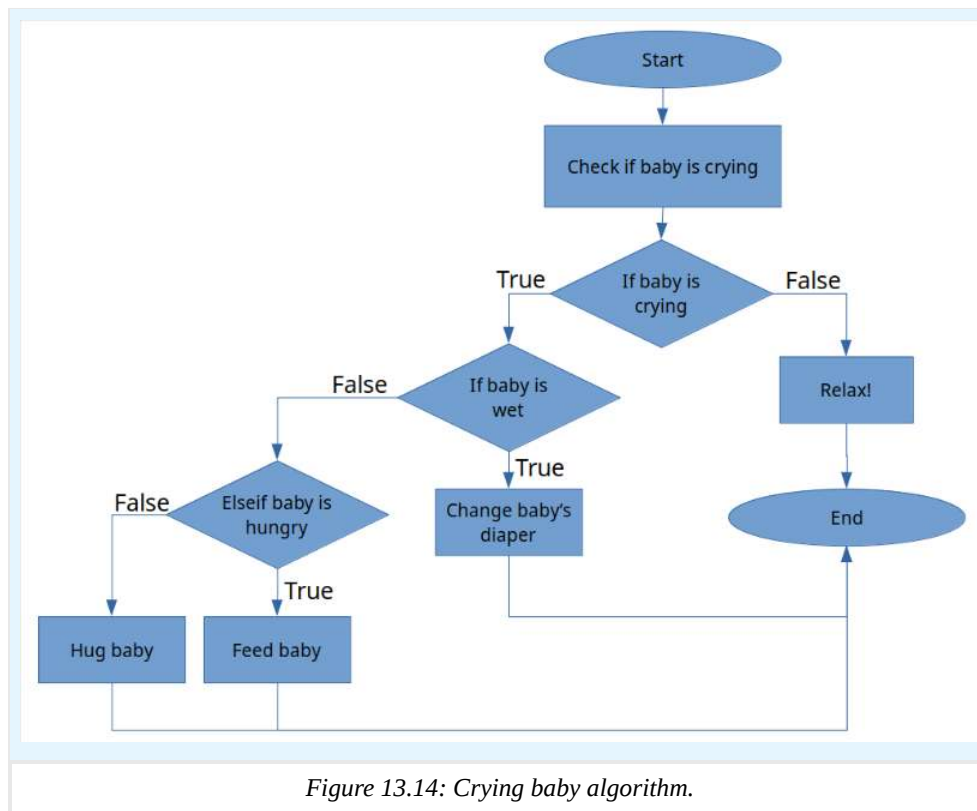
Nested Decisions

When designing your programs you may run into scenarios where it is useful to create nested decisions. Nested decisions describe the scenario of enclosing one decision into another. It may not seem intuitively obvious why this is necessary, but as your computer programs become more complex, it will be an extremely effective and useful strategy to consider.



Figure 13.13: Nested decisions allow for some neat program capabilities!

For example, consider the following flowchart, which describes an algorithm a new mother might use to stop her baby from crying:



Notice how the “If the baby is wet” condition is *nested* inside of the “If the baby is crying” condition. That is what we mean by nesting! There is no difference in the way that you write the code. You just can put one `if` after the other. The only little trick to nesting is remembering that each `if` needs a corresponding `end`. When you have several nested decisions, it can get

confusing which `end` keyword is referring to which `if` keyword. To help organize your code, MATLAB will automatically apply a tab after each `if` statement. That way you can see what “level” you are in.

For example, take a look at the code in figure 13.15 below. Keep in mind **I do not expect you to understand what is happening in this code!** (Spoiler: it is actually non-sensical code because I had to change some things to `if` statements so it didn’t include programming concepts you haven’t seen yet but that is not the point).

```
1  if iter == (size_A-1)
2      column_to_examine = U(iter:size_A,iter);
3      [~,max_loc] = max(abs(column_to_examine));
4
5      if max_loc ~= 1
6
7          max_loc = max_loc + iter - 1;
8          top_row_loc = iter;
9
10         U = rowSwap(U,max_loc,top_row_loc);
11         P = rowSwap(P,max_loc,top_row_loc);
12
13         if iter > 1
14             L_left_bound_swap = 1;
15             L_right_bound_swap = iter - 1;
16
17             L = partialRowSwap(L,max_loc,top_row_loc,...
18                 L_left_bound_swap,L_right_bound_swap);
19         end
20     end
21
22     if row_num == (iter+1)
23         mult_factor = U(row_num,iter)./U(iter,iter);
24         L(row_num,iter) = mult_factor;
25         U(row_num,:) = U(row_num,:) - mult_factor.*U(iter,:);
26     end
27
28 end
29
```

Figure 13.15: Example of non-sensical code of nesting.

You should be able to create a flow chart of this code from figure 13.15 alone. The key is to notice that on line 5, there is a *nested* `if` statement. Furthermore, there is a 2nd level *nested* `if` statement on line 13. The other thing that you should notice is that MATLAB has automatically applied a tab after each `if` statement.

Go ahead and type in some non-sensical code into MATLAB to see this auto-tab in action. Or you can just copy the code in figure 13.15.

Smart Indent

If you ever mess up your tabbing, don’t worry! Just select all your text, right-click, and select the “Smart Indent” option (see figure 13.16 below). Again, this is something that you should try yourself.

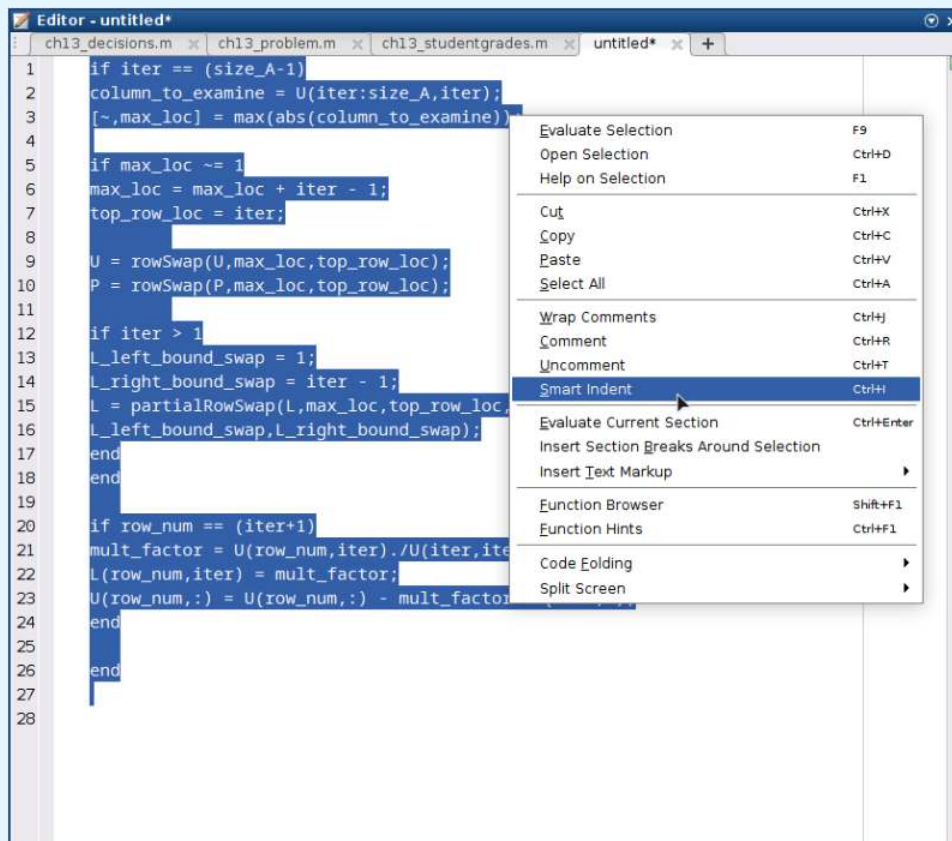


Figure 13.16: Selecting text, right-clicking, and then selecting “Smart Indent” will fix any indentation problems your code has and will make it easier to read!

This page titled [4.2.1: MATLAB Decisions](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 13: MATLAB Decisions](#) is licensed [CC BY-NC 4.0](#).

4.3: MATLAB Loops



Figure 14.1: A little different but perhaps equally as terrifying of a loop!

There are over 2.38 billion active Facebook users every month. Somehow, Facebook can send a “Happy Birthday” message to every single person that has Facebook. When you think about it, that is a remarkable feat!

Assuming an even distribution of Facebook users means that there are over 6,520,547 birthdays every day!

$$\frac{2,380,000,000 \text{ users}}{365 \text{ days/year}} = 6,520,547.95 \text{ birthdays / day} \quad (4.3.1)$$

For the sake of argument let’s say that Facebook messages everyone saying “Happy Birthday! I just wanted to tell you Happy Birthday because I am a corporation that relies on you thinking of me as a human being!” That is 26 words.

The world record for typing speed is 216 words per minute. It would take the fastest person in the world 0.12 minutes or 7.22 seconds to type that message. Not bad!

$$\frac{26 \text{ words/message}}{216 \text{ words/minute}} = 0.12 \text{ minutes/message} * (60 \text{ sec/min}) = 7.22 \text{ sec} \quad (4.3.2)$$

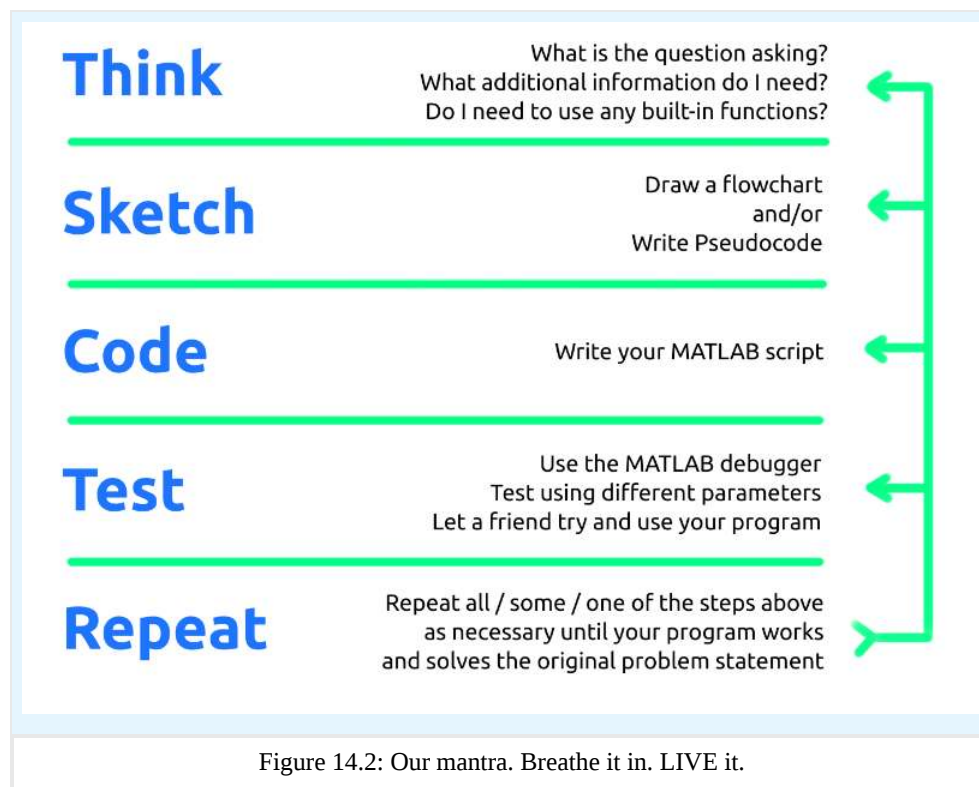
So if Facebook hired the *fastest typist in the world*, and locked them in a room and had them type that message to everyone in the world it would take them 544 days to write a short message to every person who had a birthday every day.

$$\begin{aligned} & 7.22 \text{ sec/message} * 6,520,547 \text{ messages} \\ & = 47,078,349 \text{ seconds} * (1 \text{ hour}/3600 \text{ seconds}) \\ & = 13,077 \text{ hours} * (1 \text{ day}/24 \text{ hours}) = 544 \text{ days} \end{aligned} \quad (4.3.3)$$

Clearly, this isn’t how Facebook does it. Not only is it extremely tedious, but it is also impossible. What they probably do, is have a single automated message and then loop through their database and send everybody that message who is having a birthday.

Whenever you run into a task that sounds tedious, it is likely a candidate for a computer! And you will probably need loops! So let’s learn about them.

Remember our mantra and guiding process for writing complex programs is: **think, sketch, code, test, repeat.**



Learning Goals

In this chapter, we will learn how to program loops in MATLAB. That means learning:

- What a MATLAB `for` loop is and how it works.
- How to represent a `for` loop in a flowchart.
- The only case when it is appropriate to use a `while` loop and why it is dangerous.

Maclaurin Expansion of $\sin(x)$ - A Tedious Example

Let's start this discussion by looking at a real-world example (this one comes from the world of numerical methods, a crucial mechanical engineering course you will soon take!). We will be using this one particular example for a considerable percentage of this chapter.

If you are unfamiliar with [Maclaurin expansion \(it is a special form of a Taylor series expansion\)](#) that is OK! For now, you should just understand that **it is a way to approximate a function as a polynomial with an infinite number of terms**. [Khan academy has a great explanation if you would like to understand more \(or if you need a quick refresher on infinite sums\)](#). If you are shaky on what this paragraph is saying, be sure to take a minute to check out those links and get up to speed.

For our purposes we will just understand that the following is true:

$$\sin(x) = \sum_{k=0}^{\infty} \frac{-1^k x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (4.3.4)$$

Recall, this is saying that we can represent $\sin(x)$ as an infinite polynomial and it will equal $\sin(x)$ exactly as long as we have an ∞ number of terms.

Let's consider the case where we need to approximate $\sin(x)$ with MATLAB and we decide to use the Maclaurin expansion to do so. Clearly, we can't do an ∞ number of terms, that would take an ∞ amount of time! Instead, let's truncate the expansion to 3

terms and see if it is good enough.

$$\sin(x) \approx \sum_{k=0}^{k=2} \frac{-1^k x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} \quad (4.3.5)$$

We can now use this approximation to estimate something like $\sin(2.618)$. Note, the angle in this case, (2.618) , is in radians.

Before we move on...

In order to check our approximation of $\sin(2.618)$ you first need to calculate the analytical value (i.e. true value) of $\sin(2.618)$.

Now we know the *correct answer*. We are ready to calculate our *approximation* using a subset of the Maclaurin expansion.

$$\sin(2.618) \approx 2.618 - \frac{2.618^3}{3!} + \frac{2.618^5}{5!} = 0.6523 \quad (4.3.6)$$

Ok, that isn't a terrible approximation but it isn't very good. What happens if we increase up to seven terms?

$$\sin(x) \approx \sum_{k=0}^{k=7} \frac{-1^k x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} \quad (4.3.7)$$

Well, we can be sure that it will be more accurate but it is clear that the more terms that we add, the more tedious this process becomes. Imagine increasing our estimate to 20 terms! Whenever a process is tedious and repetitive, it is a perfect candidate to have a computer do it. In order for a computer to do it though, **we need to transfer this process into an algorithm that can solve this problem for an arbitrary (n) number of terms.**

Remember our mantra, **think, sketch, code, test, repeat.**

THINK - You need to stop and **think!** You should be asking yourself these types of questions (and more!):

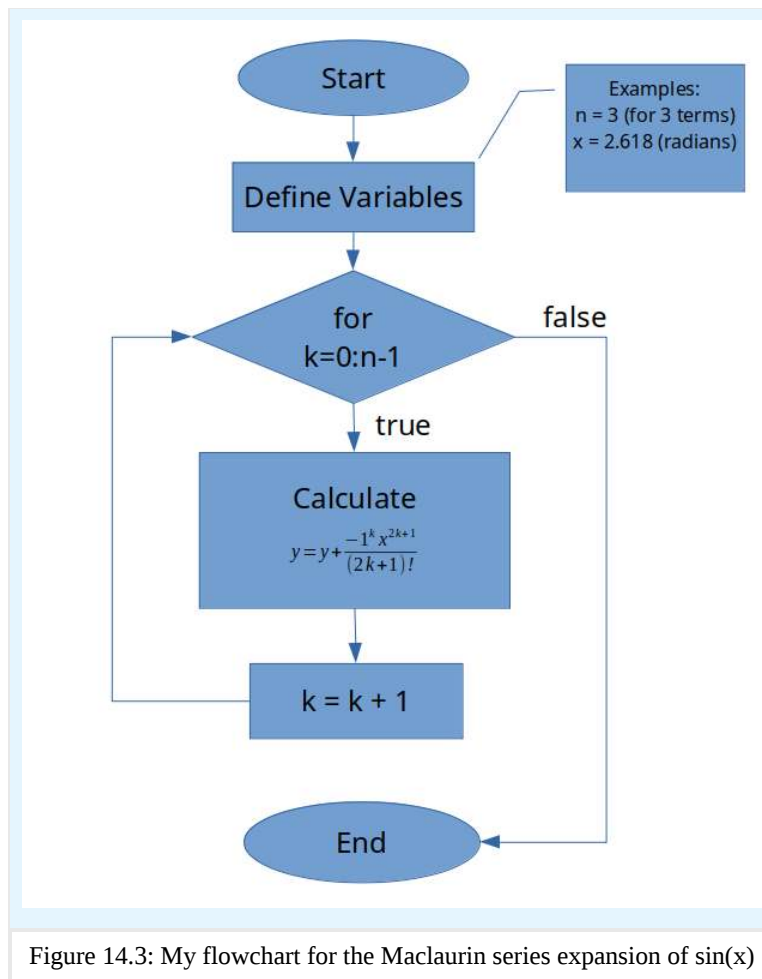
- Do you understand the expansion?
- Do you understand our goal?
- Do you have an idea for a way to calculate this in MATLAB?
- What does an (n) number of terms mean?

SKETCH - I will trust that you spent a minute thinking about the problem and making sure that you understand all the moving pieces. You wouldn't skip a brain workout, would you? The next step is to **sketch**.

Try It!

Before you move on and look at my flowchart, can you create pseudocode or draw a flowchart that can algorithmically solve this problem for an arbitrary (n) number of terms? It might be good to review what we learned in the thinking algorithmically chapter if you have not done so already.

Keep in mind that it is OK if you are confused or stuck **as long as you gave it an honest try!** Here is my flowchart for this problem:



Now is a good time to go back to the **think** step. Look at my flowchart in figure 14.3 above and see if you can answer the following questions.

- Does the flowchart make sense to you?
- Do you understand how it works? Can you follow it along?
- Do you understand what the true and false text indicators are referring to?
- Do you understand why I have the little extra explanation box giving *examples* of variables?

We have now successfully completed the **think** and **sketch** stages of our process. In order to continue with the next step, **code**, we need to learn how to program these types of structures into MATLAB. Let's take a break from this particular problem and investigate how to program the necessary functionality into MATLAB.

MATLAB For-End Loops

From our flowchart in figure 14.3 above, we can see that we are going to use a `for` loop. In the thinking algorithmically chapter we learned how and when to use a `for` loop. Now we just need to see how to program this type of loop into MATLAB.

Recall that `for` loops execute a pre-determined number of times. Generically, in MATLAB, they are programmed as follows:

```
for index_variable = first_pass : increment : last_pass

    < code to be executed during the loop >

end
```

for-end Loop Important Details:

- for loops *require* 2 numbers to be defined: `first_pass` and `last_pass` .
- The number `increment` is optional. If it is omitted, the increment will default to 1.
- MATLAB will modify the `index_variable` by the `increment` amount specified at the *start* of each pass through the loop.
- The loop is completed when the `index_variable` value is *greater than or equal to* the `last_pass` number. In the flowchart below this is what is meant by the true/false condition on the `for` loop diamond box.
- The `index_variable` persists in the workspace after the loop is completed. That means you can check it to make sure that it worked the way that you expected it to.
- The `index_variable` can either be used in calculations inside of the loop if necessary or desired.
- Similarly to decision structures, the `for` and `end` lines of code should not end in a `;`
- The `first_pass` , `increment` , and `last_pass` variables can actually contain functions or mathematical statements as long as they evaluate to a number.
- For example, consider an array named `gamma` is defined in the workspace: `for kappa = 1:length(gamma)` is valid because when the function `length(gamma)` is evaluated by MATLAB, an integer is returned. In this case, the increment is equal to 1 because a specific increment was omitted. So if `length(gamma) = 5` , this loop would run 5 times. The flowchart for the generic for loop is shown below in figure 14.4.

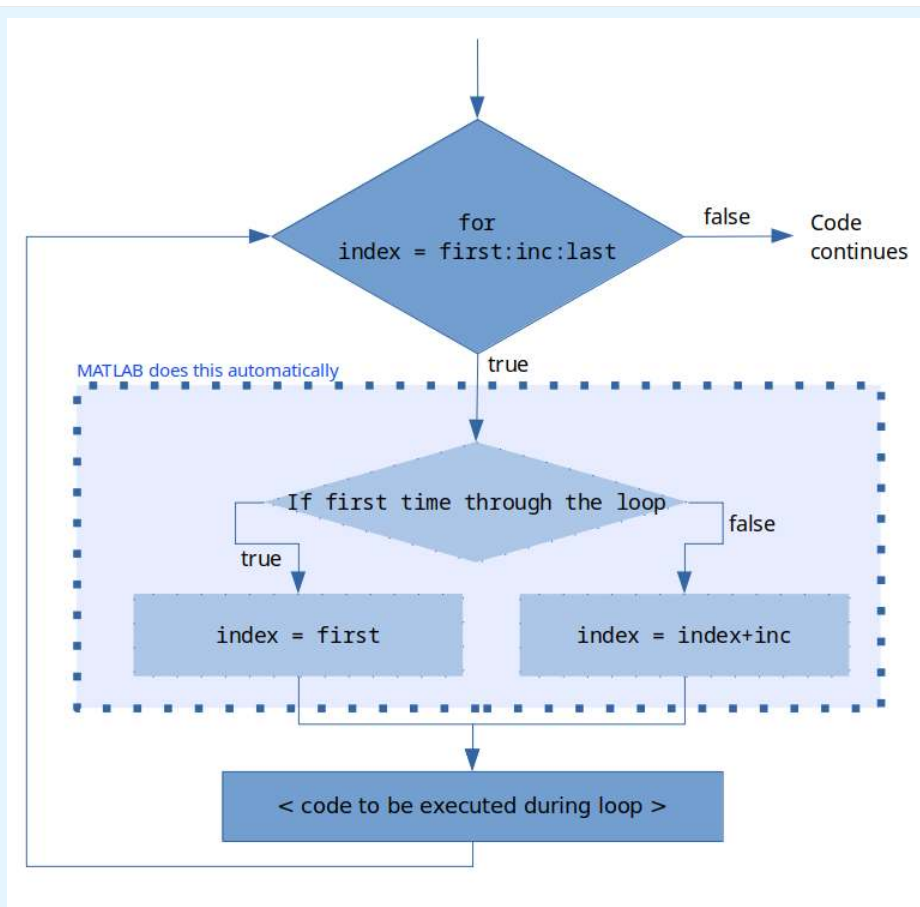


Figure 14.4: Generic MATLAB **for** loop flowchart. The variable names are a little different than in the generic code example above (e.g. **index_variable** was shortened to **index**, etc) to preserve space but the logic is the same.

As an example, consider the example code below in figure 14.5. This may look complicated but in practice, it *looks* more complicated than it really is. See if you can follow the logic presented in the figure. It would also help to try and “follow along” on your own on a sheet of paper. See if you can follow the `value` variable throughout the looping process.

Example Code

```
value = 0;

for z = 1:2:5
    value = value + z;
end
```

In this example
index_variable is z
z = 1 on first pass
increment = 2
z = 5 on last pass

How MATLAB Runs the Example Code

value = 0

Recall: MATLAB runs code
top to bottom

Enter Loop

Initialized when for
loop starts

z = 1
value = 0 + 1 = 1

Executing code inside
loop. value = 0 and z = 1

Re-enter loop

Because z < last_pass

z = 1 + 2 = 3
value = 1 + 3 = 4
z = 3 + 2 = 5

Once you enter loop:
z = z + increment

Re-enter loop

Because z == last_pass
Final Variable Values
value = 9
z = 5

value = 4 + 5 = 9
z = 3 + 2 = 5

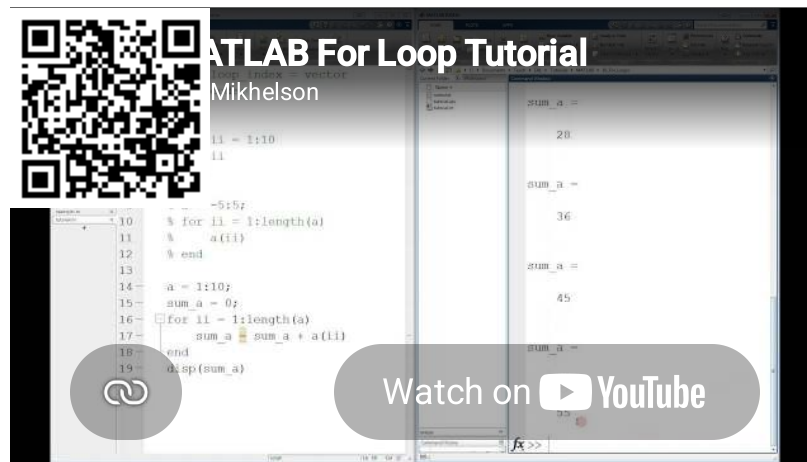
Code Complete

Figure 14.5: An example of a for loop execution process. You can confirm this by executing the sample code in MATLAB on your own!

Stop and Think

You need to spend some time reviewing figures 14.4 and 14.5 to really understand how these loops work. If this explanation is confusing to you, try taking a break from this text, and looking at someone else's explanation. The important thing is that you understand what `for` loops are and how to use them.

If you search for "for loops" on YouTube you can find hundreds of videos of people explaining these looping structures. [Here is one such video.](#)



for-end Key Concept #1 - Loop Executed a Specified Number of Times

In the example loop: `for z = 1:2:5` we can see that this loop will only run a total of three times. The first time through the loop `z = 1`, the second time through, `z = 3`, and the last time through, `z = 5`. The key to remember is that the for-end loop will only run a specified number of times that is pre-determined based on the first line of the loop.

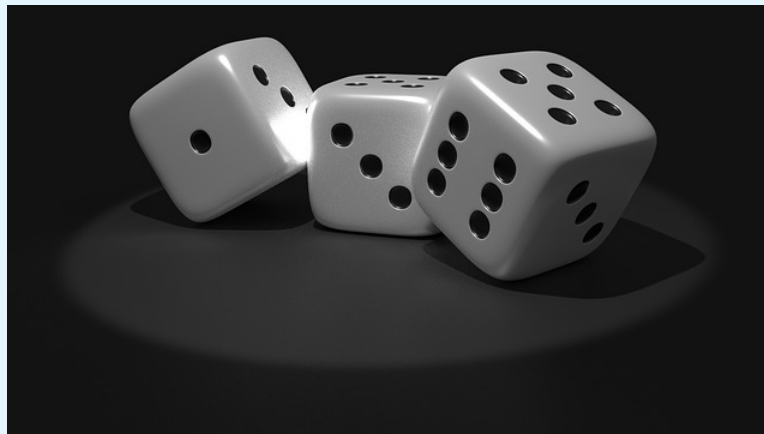


Figure 14.6: The loop will only run a specified number of times. Unlike the roll of the dice, this is predetermined by the programmer.

When designing your programs, it is important to think about how to define your `index_variable` and what values to specify for it so that you can control the number of passes through the loop.

for-end Key Concept #2 - Follow the Variables

In addition to predicting the number of times loop code will execute you should also be able to “follow the variables” through the execution of the code. Following the variables is important so that you can troubleshoot errors in your loops and predict what values should be generated by your loop. An example of following the variables was shown in figure 14.5 above. The idea is that you can **sketch** what should happen when the code has been run.

for-end Key Concept #3 - Analyze Arrays

The final important thing to consider when learning about `for-end` loops is that they can be used to analyze and build arrays. For example, let's consider that we have an array defined in the workspace as follows:

```
man_bear_pig = [1, 1, 2, 1, 4, 2, 4, 5];
```

As an example, let's say that we want to add all the values in this array (you can use the built-in `sum()` function for this but let's consider how to do this using loops as an example). We can use a for loop to cycle through all of the values in the array! This code will accomplish this:

```
1 sum = 0;
2
3 man_bear_pig = [1, 1, 2, 1, 4, 2, 4, 5];
4
5 for index = 1: length(man_bear_pig)
6
7     sum = sum + man_bear_pig(index);
8
9 end
```

runrun allrestartrestart & run all

The key here is to notice how the `for` loop was used to loop through every value of the array `man_bear_pig`. Since the increment is 1, we can use the `index` variable to access the data within the array. This is a common application of for loops and one that you need to have in your tool belt to be a successful programmer.

For this problem, consider the following code:

```
1 sum = 0;
2
3 man_bear_pig = [1, 1, 2, 1, 4, 2, 4, 5];
4
5 for index = 1: 2: length(man_bear_pig)
6
7     sum = sum + man_bear_pig(index);
8
9 end
```

run

run all

restart

restart & run all

Finishing the Maclaurin Example

Hopefully, at this point, you can at least understand how a `for-end` loop is supposed to work and what it might be used for. You should also start to understand the 3 key concepts you should learn about these loops. It is OK if you are a little shaky in your knowledge, but you should at least know what you need to learn.

Try It!

Returning to our Maclaurin series example, we are *now* ready for the **code** phase. Based on what you have learned about `for` loops, can you write the MATLAB code that accomplishes our programming goal? To remind you, the goal was:

Create a MATLAB script that can return the value of the Maclaurin series expansion of $\sin(x)$ for angle x in radians, for an arbitrary n number of terms.

- Recall the Maclaurin series expansion of $\sin(x)$ is:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (4.3.8)$$

- Remember that although only 4 terms are shown in the bullet point above, that this is an ∞ sum
- When we say an arbitrary n number of terms, your script should allow the user to change a variable `n` to change how many terms are used in the estimation.
- For example, if the user specifies `n = 3`, and wants to estimate $\sin(1.15)$ by specifying `x = 1.15`, the program should calculate: (this is actually a pretty good estimate by the way) Give it your best shot! Try and see if you can come up with this code on your own! I believe in you!

```
1 % Insert your Maclaurin code here
```

run run all restart restart & run all

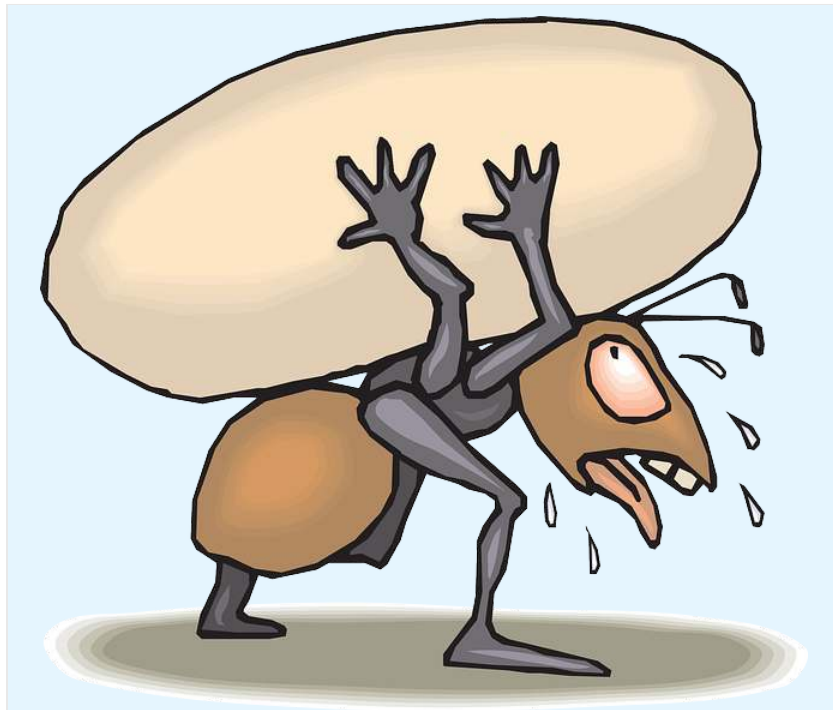


Figure 14.7: I know this is difficult but no pain no gain! Give it your best shot!

My solution is shown below in figure 14.8:

```

1 % sin_maclaurin.m
2 % Created on 1 June 2019
3 % By: Samuel Bechara, PhD
4 % Description: calculate sin(x) by using Maclaurin series
5
6 % angle of interest in radians
7
8 x = 1.15;
9
10 %m minimum 2 terms
11
12 num_terms = 2;
13
14 % initialize output
15
16 y = 0;
17
18 % loop through terms adding each time
19
20 for index = 0:num_terms-1
21     y = y + (-1)^index*x^(2*index+1)/factorial(2*index+1);
22 end

```

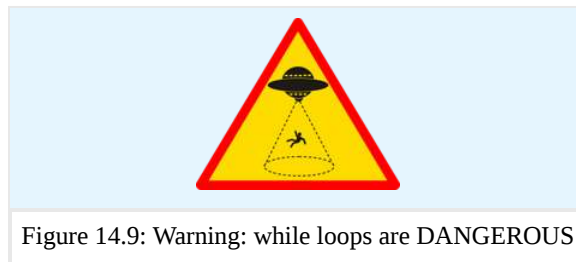
run run all restart restart & run all

Figure 14.8: My solution for the Maclaurin expansion of $\sin(x)$

If you didn't get this exactly it is OK as long as you tried. Spend some time with the script above in figure 14.8 until you understand how it works. Keep practicing, reading, and watching YouTube video tutorials until this is starting to click!

The Dangerous Loop

This chapter would not be complete without a mention of `while-end` loops. They will be described below, however, I encourage you to *only* use `while-end` loops in cases where it is impossible to use `for-end` loops.



The reason I am warning about while-end loops before discussing them is that they are the only thing that we will learn in this book that can create ∞ loops.

What is an ∞ loop? Well, why don't you go ahead and try one and see?

How to Kill (and Create) an Infinite Loop

Before I show you how to create an infinite loop, I will first tell you how to kill one.

To stop an infinite loop, select the MATLAB command window and hit `ctrl+c` on your keyboard. OK, ready to create an infinite loop? Create the following script, save it as `to_infinity_and_beyond.m`, and click run. Once you get bored of seeing the numbers fly by, hit `Ctrl+c` on your keyboard to stop it.

```
1  % to_infinity_and_beyond.m
2  % creates an infinite loop, remember Ctrl+c stops it!
3
4  z = 0;
5
6  while z >= 0
7      z = z + 1 % leave out the ; so we can see numbers counting up
8  end
9
```

Figure 14.10: To infinity, AND BEYOND! I hope Disney doesn't sue me...

What was going on there? The reason that `while` loops are dangerous is because they *do not have a predetermined number of times that they run*. Instead, while loops run until the condition they were initialized with is false. The reason this loop keeps running forever is that the initialized condition, `z >= 0` was never made false. Since `z` started equal to `0`, and all we did in the loop was keep adding positive values to it, `z >= 0` was *always* true.

MATLAB while-end Loops

So what are `while` loops and what do they look like? The generic `while-end` loop is shown below:

```
while < conditional expression is true >  
  
    < execute code >  
  
    < * YOU BETTER MAKE THAT CONDITION FALSE! * >  
  
end
```

`while` Loop Important Details:

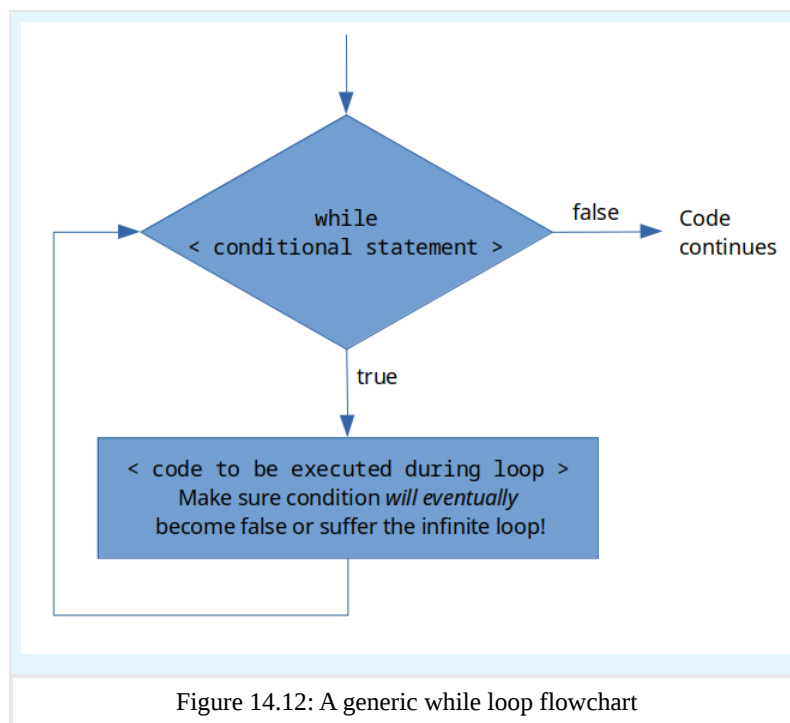
- Similar to `for` loops, `while` loops start with the keyword `while` and must be completed with the `end` keyword.
- It is vitally important that your `while` loop conditional expression is made false at some point during the execution of the loop. If it is not made false, you get an infinite loop.

You can think of `while` loops as over-eager dogs. They will run themselves to death unless you tell them to stop. The way that you tell a while loop to stop is to make sure that at some point in the loop, the initial conditional expression becomes false.



Figure 14.11: The while loop is the overenthusiastic sled dog of the loop world. [Sled dogs do actually run themselves to death.](#)

The flowchart for the generic `while` loop is shown below in figure 14.12:



The beauty of `while` loops is that you can ensure that your code repeats itself until a specific condition is met. However, `while` loops should only be used in instances where you do not already know how many times the loop needs to be run.

Maclaurin Series Example - `while` Loop Edition

To see an instance where a `while` loop is necessary, let's consider the case of the Maclaurin series expansion of $\sin(x)$ one last time. In this case, let's flip the question a little bit. Let's say that the new question is, **you want to know how many terms of the Maclaurin series are necessary to get an estimation of $\sin(x)$ that is 99% accurate.**

We know from the Maclaurin series that if we keep adding terms, the estimation will get more and more accurate. We also know that we need an infinite number of terms to make the answer infinitely accurate. But how many terms are necessary to get our estimation of $\sin(x)$ to be 99% accurate? It will depend on the x that the user specifies but it actually might not take as many terms as you might think it does.

Here are a few tips to get you started:

- It is actually easier to use the error rather than accuracy. The formula for percent error is $(\text{true} - \text{approximate}) / \text{true} * 100$. So 99% accurate would be less than 1% error.
- Use the built-in MATLAB function `sin()` to get a “true” value for $\sin(x)$. Even though it isn't 100% accurate, it is close enough for our purposes.
- Since you do not know how many terms it will take, you will need to use a `while` loop.

Try It!

I am not going to lie, I think that this is a pretty difficult problem. But we are here to work out our brains and difficult problems are our opportunity to show off our strength!

Try to complete the entire process: **think, sketch, code, test, and repeat.**

- **Think** - what is this problem asking for? Do you have anything already completed that can help you complete this?

- **Sketch** - Write out a flowchart or pseudocode for this problem and how you would solve it.
- **Code** - Give it your best shot! Make sure that you **repeat** the **think** and **sketch** phases as much as you need to!
- **Test** - Compare your MATLAB solution to your solution by hand (it won't take as many terms as you think!). Try with different x values and make sure that it is robust and it works.
- **Repeat** - repeat any steps necessary until you can get a working script. I know this is difficult! You can do it!

```
1 % Here is a code box to enter your Maclaurin while loop code
```

runrun allrestartrestart & run all

The code solution to the problem is shown below in figure 14.13. Make sure that you give it an honest try before looking at the solution though! Do not skip your brain workouts, you will only cheat yourself.

```
1 % sin_maclaurin_num_terms.m
2 % Created on: 3 June 2019
3 % By: Samuel Bechara
4 % Description: estimate the number of terms (var, num_terms) it takes to get the Maclaurin series expansion
5
6 % angle of interest in radians
7
8 x = 2;
9
10 % since the first term is just x, start there
11
12 est = x;
13 index = 0;
14
15 % assume MATLAB's sin() function is perfect, it isn't, but it's close
16
17 actual = sin(x);
18
19 % loop through, checking error percentage each time
20
21 while abs((est-actual)/actual)*100 > 1
22     index = index + 1;
23     est = est + (-1)^index * x^(2*index+1)/factorial(2*index+1);
24 end
```

runrun allrestartrestart & run all

Figure 14.13: The solution. The **num_terms** variable will be an integer that is equal to the number of terms necessary to get better than 99% accuracy.

A Final Note on Loops

Look, I get it. Programming is hard. Loops are hard. But remember that we are treating this as a brain workout. No one walks into a gym and expects to bench 500lbs right away! You need to work out for years before you get to that level. The good thing about programming, you do not need to practice for years to understand loops! *You will need to practice though.*

The other thing to remember is that not everyone is equal. Just like working out, some of your classmates or peers may get this faster than you. It doesn't matter. The only thing that matters is YOU. You need to take the time necessary to understand these concepts. I believe in you!

Challenge Question - Return to the Student Grade Analyzer

Speaking of practice, here is one last challenge question to end the chapter with. Remember the last chapter when we used decisions to assign a letter grade to a numeric grade? Now consider the case where instead of a single numeric grade, you instead have an array of grades. Let's say that we have an array called `grades` that is a column vector that contains the final grades for everyone in a class. When you are working on this, only consider 10 or so grades but understand that your algorithm should work for any number of grades (my classes can often be 200+ students!).

Remember, the final grades are calculated as follows:

Grade Range	Letter Grade
grade > 90	A
80 < grade < 90	B
70 < grade < 80	C
grade < 70	F

Table 14.1: How letter grades are calculated from numeric grades

Can you create a MATLAB script that:

- Accepts a 1D row vector called `grade` that contains numeric grades.
- `grade` can have *any number* of grade entries.
- Upon completion, your script creates a corresponding 1D row vector called `letter_grade` array. The index of the arrays matches, so if `grades(2) = 92`, `letter_grades(2) = 'A'`

```
1 % Write your challenge code here
```

run

run all

restart

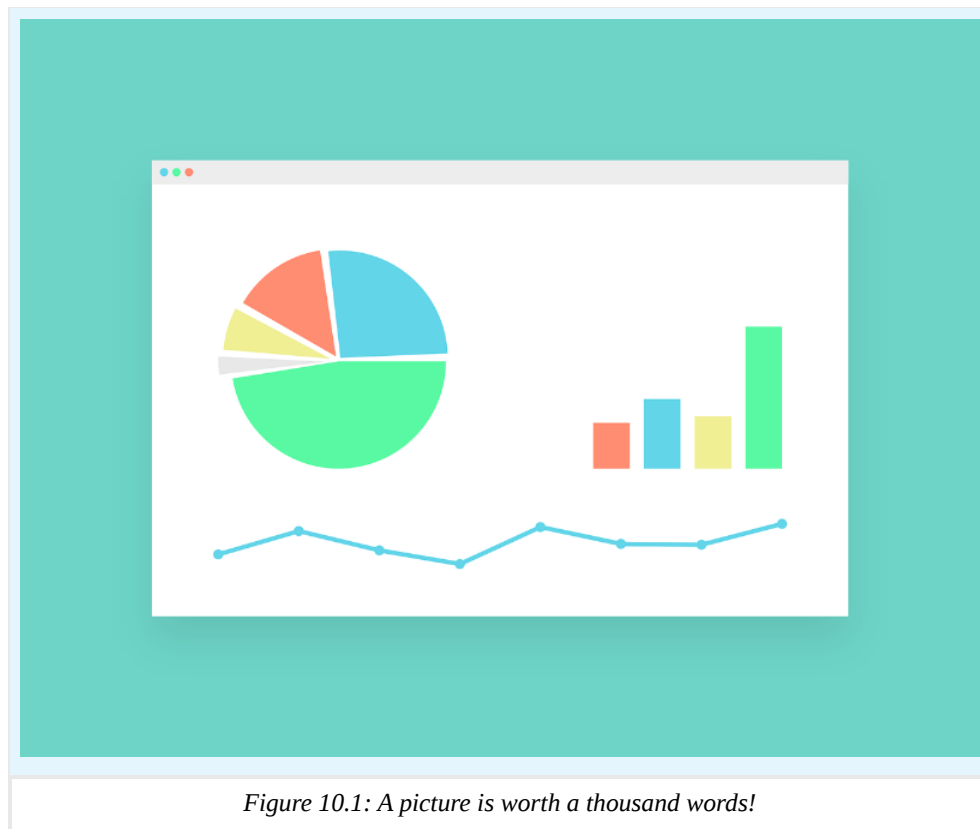
restart & run all

Good luck practicing! Remember our mantra and guiding process for writing complex programs is: **think, sketch, code, test, repeat.**

This page titled [4.3: MATLAB Loops](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 14: MATLAB Loops](#) is licensed [CC BY-NC 4.0](#).

4.4: MATLAB Plotting



Remember, most cliches are cliches because there is a hint of truth in them. For example, the adage “a picture is worth a thousand words” is persistent because graphs and plots are extremely useful at portraying information in a compact manner. MATLAB has robust plotting and graphing capabilities and can generate plots in very few lines of code. Unlike Excel, there is very little in terms of buttons or menus that you have to navigate. Instead with MATLAB, you need to remember a few lines of code but in the end, it is much faster and much more flexible in generating your plots.

As I did in Chapter 8, **I highly recommend that you create a new script** called `chapter10_examples.m` and save it somewhere that you will remember. When you are following along with this chapter, make notes and comments to that script so that you can refer to it later and what you have learned.

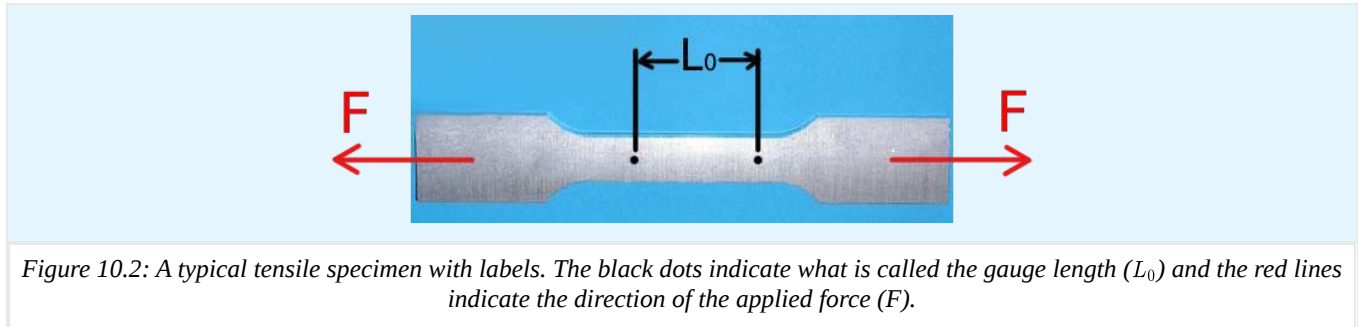
Learning Goals

In this chapter, we will learn how to create good plots in MATLAB. We will remember everything we learned in Chapter 5 and in addition will learn:

- How to use what we have already learned about arrays to streamline making plots in MATLAB.
- How to appropriately label axes, titles, and add legends to our plots.
- What does `hold on` and `hold off` mean and when and how to use them.
- How to format our plots with different colors, line styles, and other advanced formatting options.
- Subplots
- Histograms

Let's Make a Plot!

Remember, the philosophy of this book is that diving in and getting your hands dirty (metaphorically at least) is good for learning! So before we dive into too many details about plotting, let's consider the case of a real engineering application, the tension test. Don't worry, we won't dive into too many of the details as you will learn all this in a later class, just the basics so we do not have to work on a contrived example.



The goal of a tensile test is usually to determine the materials [elastic modulus](#).

When the tensile specimen is pulled, the material will deform and the initial L_0 will increase to a new length L . We call this **engineering strain** and it is defined as:

$$\epsilon_e = \frac{L - L_0}{L_0} \quad (4.4.1)$$

Note that the units for engineering strain are Length / Length which means it is a unit-less quantity! It is just a ratio of the change in the length to the original length of the specimen.

The force used to deform the specimen divided by the specimen's cross-sectional area is called the **engineering stress**:

$$\sigma_e = \frac{F}{A_0} \quad (4.4.2)$$

You can probably intuit that as the force keeps increasing, the specimen keeps getting pulled further and further apart, and eventually, it will break. But before it breaks if you let go of the force, the material will return to its original shape and size. A property of the material (before it breaks) called elastic modulus can be determined by the ratio of the stress and strain:

$$E = \frac{\sigma_e}{\epsilon_e} \quad (4.4.3)$$

Since we determined that the units of stress were Pascals, and strain is unitless, the units of elastic modulus are also Pascals.

There is a lot more to learn about elastic modulus and tensile testing but this information should give you enough knowledge to start!

Stop & Think

Look at the equation for elastic modulus again. Can you figure out why it is useful? Elastic modulus gives the engineer an idea of how much the material resists being deformed. Can you understand that from the equation?

Now that we have that out of the way, let's do a real-world example of how to plot in MATLAB!

Let's consider a specimen made of high-density polyethylene (HDPE). The specimen has a round cross-section with a radius of 5 mm (this implies that the cross-sectional area is $7.854 \times 10^{-5}\text{meters}^2$, make sure you can calculate this on your own! It is good units practice). Consider the following Force N and gauge length m data from a tensile test experiment shown below in table 10.1.

Force (N)	Length (m)
0	0.025
5000	0.0270
10000	0.0290
15000	0.0311
20000	0.0329
25000	0.0350
30000	0.0370
35000	0.0390
40000	0.0408

Table 10.1: The results from a tensile test experiment.

Try It!

Enter the values of Force in as a vector `force`, and enter the values for the length in as a vector variable `len` (Don't use the variable name 'length' because that will overwrite the built-in function that we learned about the last chapter.) in a new script. Now that we have these in, we need to create vectors called `e_stress` and `e_strain` that correspond to the engineering stress and engineering strain respectively. Can you do this on your own? Try before reading on below! Don't skip your brain workout!

```
1 % Try It 10.1
2 % Create a "force" and "len" variable using the data above.
3
4 force =
5
6 len =
7
8 % Calculate the stress and strain variables based on the equation listed earlier in the chapter.
9
10 e_stress =
11
12 e_strain =
```

run

run all

restart

restart & run all

Your script should look very similar to mine shown below in figure 10.3:

```
1 %% Chapter 10 Follow Along Work
2
3 % Define the force vector
4
5 force = 0:5000:40000;
6
7 % Define the length vector
8
9 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
10
11 %Calculate area based off information
12
13 area = pi*0.005^2;
14
15 len_0 = 0.025; %initail length
16
17 %Calculate stress and strain according to equations
18
19 e_stress = force / area;
20
21 e_strain = (len-len_0)/len_0;
```

run

run all

restart

restart & run all

Figure 10.3: What your script should look like, or should at least be VERY similar. I took a shortcut on line 7 and typed in the lengths in mm then converted them to meters at the end by dividing by 1000. Sneaky!

Now to the plotting! Since we have arrays `e_strain` and `e_stress` already defined in the MATLAB workspace, we simply need to tell MATLAB to plot them! Engineers typically look at stress vs strain plots with strain on the x-axis and stress on the y-axis. To accomplish this in MATLAB, add the following line to your script:

```
>> plot(e_strain,e_stress)
```

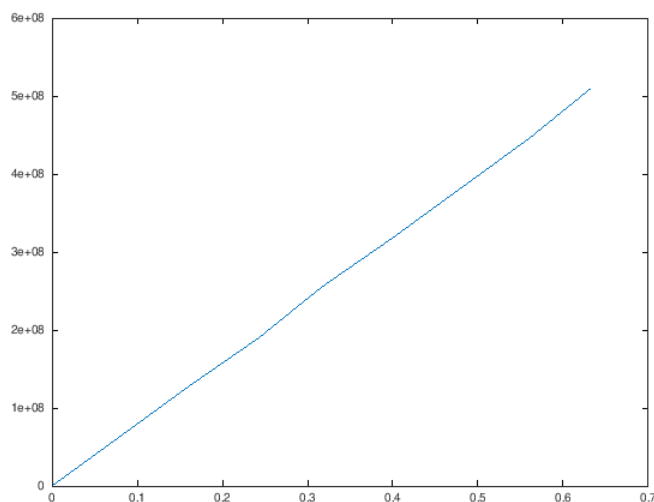


Figure 10.4: The MATLAB plot of stress vs strain.

When you hit the run button, you should see the following figure pop up (see figure 10.4 above). Congratulations! You just plotted in MATLAB! Plotting in MATLAB is really that easy. The line isn't perfectly straight because experimental data is never perfect. But here we can observe that there is a linear relationship between how much strain the specimen is under and how much it deforms! Just for fun, the slope of this line would be the elastic modulus for HDPE.

Now we just need to learn some of the details. For example, the plot in Figure 10.4 looks nice but it is missing labels, titles, and legends which we learned in Chapter X are important to have. For the next section, I will assume that you have vectors `e_stress` and `e_strain` loaded into your workspace so that you can follow along.

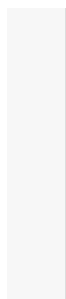
The `plot()` function

As you have just seen, MATLAB has a built-in `plot()` function that can accept single or multiple vector inputs. In our example the inputs were `e_stress` and `e_strain` but keep in mind they can be *any variable that is a vector* to generate plots.

There are lots of advanced plotting functionalities, to see a comprehensive list of available options, type `>> help plot` at the command line and read the help text for the `plot()` function. **Go ahead and take a second to skim through the help text.** Don't skip this step, it will help set up our discussion for the rest of this chapter. The most important section of the help text is displayed below in Figure 10.5. In this chapter, the three main options of the `plot()` function that we will concentrate on will be:

1. Plot Colors
2. Plot Symbols
3. Plot Lines

```
1 % The code boxes in this textbook also have the help function built in.
2 % Check it out:
3
4 help plot
```



run

run all

restart

restart & run all

Various line types, plot symbols and colors may be obtained with **plot(X,Y,S)** where S is a character string made from one element from any or all the following 3 columns:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

Figure 10.5: Example of the `plot()` help text that is most important for our Chapter.

Plot Colors

The default blue color is nice, but MATLAB includes easy-to-use commands to change the colors of your graphs. For example, let's say we wanted the same plot that we just completed but we want the line to be green. You should have noticed from the help text that there is an *optional* third input to the `plot()` function that allows us to change the plot colors.

To change the color of your plot to green, simply add the third input to the `plot()` function as a character string (A character string is surrounded by single quotes. So for the color green, the third input to the function needs to be `'g'` . So you need to change the line in your script from:

```
>> plot(e_strain,e_stress); change to >> plot(e_strain,e_stress,'g');
```

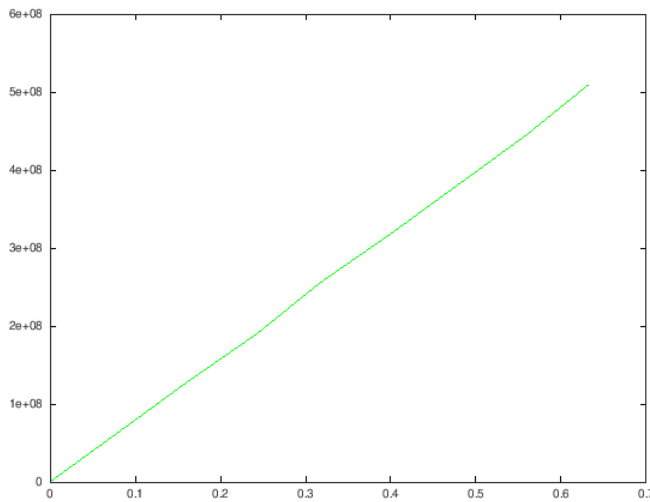


Figure 10.6: New green line, and it is so easy and fast!

When you re-run your script, you should notice that the line color has changed to green (Figure 10.6)!

Try It!

Take a moment to play around with some of the different color options to see what they look like. Note (see the help text or Figure 10.6 above) that some of the color codes are a little weird. For example, to make the line black, you need to use the character string 'k' because 'b' is reserved for blue.

```
1 % Try It 10.2
2
3 % Copy your code from the previous exercise but change the graph to plot yellow circles with a dash dot
4 % Insert code here:
```

run

run all

restart

restart & run all

Plot Symbols

When you are displaying experimental data (as we are in this example), scientists and engineers usually prefer to represent each point of data with a symbol. This visual cue tells the reader of the plot that there is no continuity between the points, instead, each point represents a distinct measurement.

So let's change our plot so that MATLAB displays each point as a magenta pentagram. Looking at the help text in Figure 10.6 above we can see that the character string, 'm' corresponds to the color magenta and that the character string 'p' corresponds to the pentagram symbol.

So to change our plot so that is showing magenta pentagrams we can change the line of our script from:

```
>> plot(e_strain,e_stress); change to >> plot(e_strain,e_stress,'mp');
```

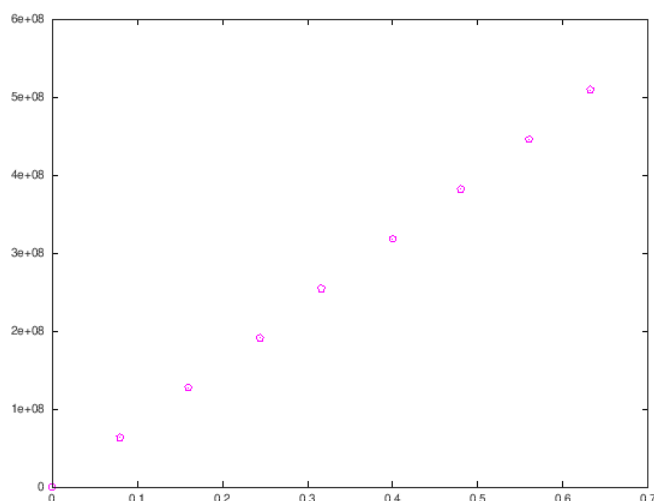


Figure 10.7: Magenta pentagrams.

Voila! Our plot is now magenta pentagrams!

Don't forget to take a moment and play around with some of the other symbol options! Notice that there are a bunch of different triangle options (triangle down, triangle up, etc). Play around with those symbols and see if you can see the difference!

Another thing to notice is that **the order of the character strings does not matter**. That means that `plot(e_strain,e_stress,'mp')` works the same as `plot(e_strain,e_stress,'pm')` !

Plot Lines

The last thing that we need to learn is how to change the plot line. Again, direct your attention to the help text for the plot function (and figure 10.7 above). Notice that we can add a character string to create solid, dotted, and other types of lines.

We can do so in the exact same way, by adding a character to the character string. So for this example, let's say we wanted **black** (remember, that is character string 'k'), **left triangle marks** connected with a **dash-dot** line. Just like before we would change the last line of our script:

```
>> plot(e_strain,e_stress); change to >> plot(e_strain,e_stress,'k-.' );
```

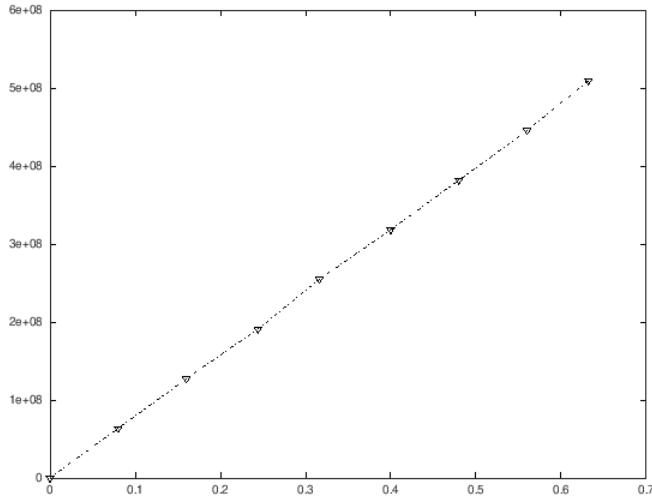


Figure 10.8: Black left arrows connected by a dash-dot line.

And that is all there is to it! Remember, the order of the character strings *does not matter*. You should also remember that you **do not have to memorize these character string codes!** Just remember that you can look them up at any time from the command window by typing in `>> help plot` .

Exercise Your Knowledge

Example Script File Up To This Point

Hopefully, you have been following along. This is what my script looks like up to this point. Notice that I commented out the plot commands I wasn't using (lines 22–24 in figure 10.9) because they will overwrite each other and it is nice to keep a record of what you did for studying later. You should also notice how my comments make my script easy to read!

```

1 %% Chapter 10 Follow Along Work
2
3 % Define the force vector
4
5 force = 0:5000:40000;
6
7 % Define the length vector
8
9 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
10
11 %Calculate area based off information
12
13 area = pi*0.005^2;
14
15 len_0 = 0.025; %initail length
16
17 %Calculate stress and strain according to equations
18
19 e_stress = force / area;
20
21 e_strain = (len-len_0)/len_0;

```

```
22
23 % plot (e_strain, e_stress) % original plot command
24 % plot (e_strain, e_stress, 'g') % plot a green line
```

run

run all

restart

restart & run all

Figure 10.9: Example script file up to this point.

Multiple Plots

A lot of the time it is useful to show someone multiple plots. Perhaps you want to compare two materials or show different projections. MATLAB has the capability to plot multiple plots on the same axes or display multiple plots in the same figure. See figure 10.10 below for an example.

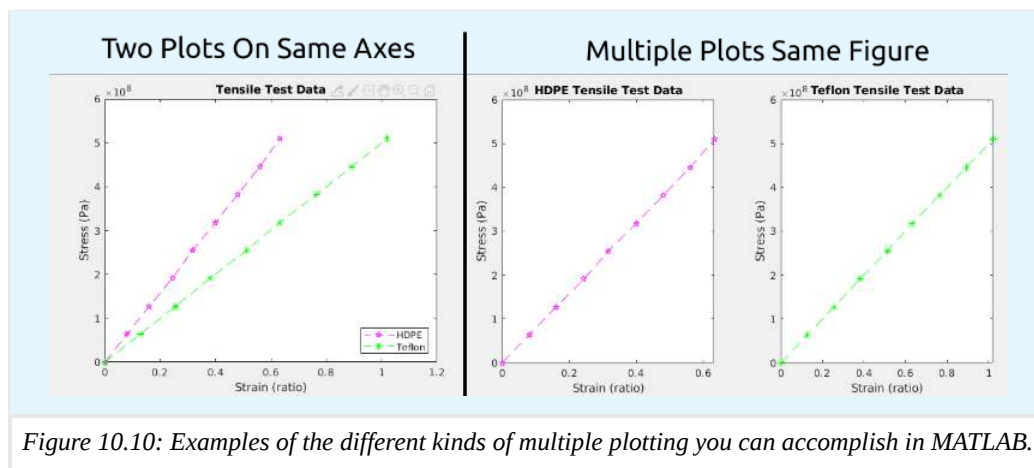


Figure 10.10: Examples of the different kinds of multiple plotting you can accomplish in MATLAB.

You should have also noticed that the plots in Figure 10.10 above have titles, axes labels, and legends (where appropriate). In the next section of this chapter, we will learn how to accomplish this. For now, let's concentrate on what we need to learn to do either:

1. Create two plots on the same axes.
2. Create multiple plots *in the same figure window*.

Try It!

Before we jump into learning those two points, you should have noticed that I added tensile testing data on Teflon to the charts. To follow along, you will need to do so as well. Go ahead and add a new array called `len_teflon` to your script with the following values:

```
len_teflon = [25 28.2 31.4 34.5 37.8 40.8 44.1 47.3 50.5]/1000;
```

We will consider the case where the Teflon specimen has the exact same starting dimensions as the HDPE specimen we are using before. We will also test the Teflon under the exact same engineering stresses. With that being said, calculate the engineering stress in an array called `e_stress_teflon` and the engineering strain in an array called `e_strain_teflon`. Do not skip ahead to see my solution UNTIL you have tried it yourself!

```
1 % Insert your code here:
2 len_teflon = [25 28.2 31.4 34.5 37.8 40.8 44.1 47.3 50.5]/1000;
3 e_stress_teflon =
4 e_strain_teflon =
```

runrun allrestartrestart & run all

To calculate the engineering stress and engineering strain in Teflon, I added the following 2 lines to my script:

```
e_stress_teflon = force / area;
```

```
e_strain_teflon = (len_teflon-len_0)/len_0;
```

Create Multiple Plots on the Same Axes

Now that we have two sets of data, we can practice plotting them on the same axes. To do so, you just need to pretend you are a cowboy, and that MATLAB is a horse. So we just need to tell MATLAB “to hold on there partner, I want to plot multiple plots on the same axes”.



Figure 10.11: Woah there, partner. Hold on.

I wasn't kidding! You really just have to add the line `hold on` to your script! For example, if we wanted to plot the HDPE and Teflon tensile test data on the same axes, we can just type in:

```

1 %% Chapter 10 Follow Along Work
2
3 % Define the force vector
4
5 force = 0:5000:40000;
6
7 % Define the length vector
8
9 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
10
11 %Calculate area based off information
12
13 area = pi*0.005^2;
14
15 len_0 = 0.025; %initial length
16
17 %Calculate stress and strain of HDPE according to equations
18
19 e_stress = force / area;
20
21 e_strain = (len-len_0)/len_0;
22
23 %Calculate stress and strain of teflon
24

```

run

run all

restart

restart & run all

The first line plots the HDPE data as we have done before (make sure you know what `'mp--'` does!). The second line (`hold on`) tells MATLAB to wait and plot subsequent plots on the same axes. After you turn `hold on`, MATLAB will put ALL plots on those axes. Therefore, it is a good idea to turn `hold off` when you are done in case you need to plot other things, not on the same axes.

Go ahead and try it in your script! Notice how it plots the two plots on the same graph!

```
1 % Try it 10.4
2 % Here is your code box to paste your code and try the "hold on" function
```

run

run all

restart

restart & run all

Creating Multiple Plots in the Same Figure Window

In this particular example (tensile test data of HDPE and Teflon) it makes the most sense to plot the graphs on the same axes. But for the sake of learning, let's go ahead and create a new figure window and create multiple plots in the same figure window. To do so, we will need to use the `subplot()` function.

subplot(m,n,p)

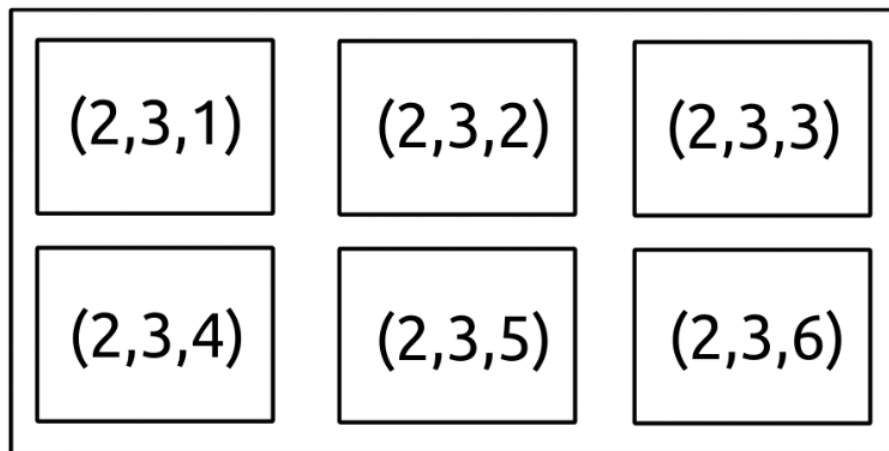


Figure 10.12: Example of how to use the `subplot()` function and what *m*, *n*, and *p* should be.

First, take a second and type in `help subplot` into the command window and read the help text.

The trick with a subplot is to follow the following steps *before you start coding*:

1. Think about how many plots you need. For this thought experiment, let's say we need 6 plots.
2. Then think about how you want your plots arranged. It really helps to sketch it out on a little piece of paper (kind of like in figure 10.12) to get a good idea. Let's say for this example we want 2 rows and 3 columns of plots.
3. After you think, you are ready to use the `subplot()` function.

The `subplot(m,n,p)` function divides the figure window into $m \times n$ rectangular plots. The `p` indicates the number of the subplot. The subplots are numbered starting at 1 in the upper left corner and increasing in number from left to right. Figure 10.12 is illustrating this. For the plot in the upper left corner, the user would type in:

```
subplot(2,3,1)
```

[insert plot command here]

Then if the user wanted to plot the next one to the left they would just continue...

```
subplot(2,3,2)
```

[insert plot command here]

And so on and so forth.

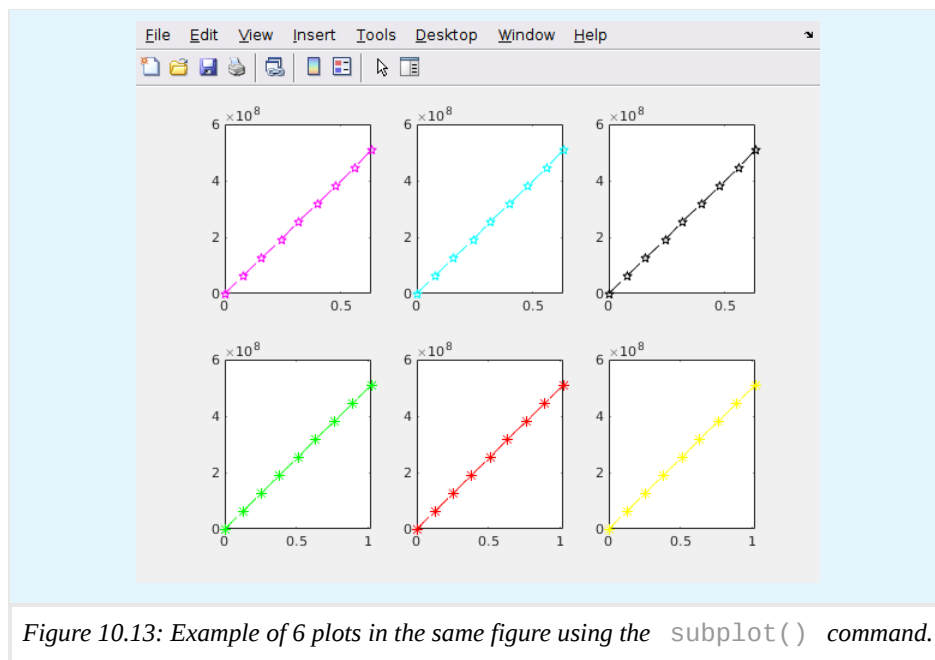
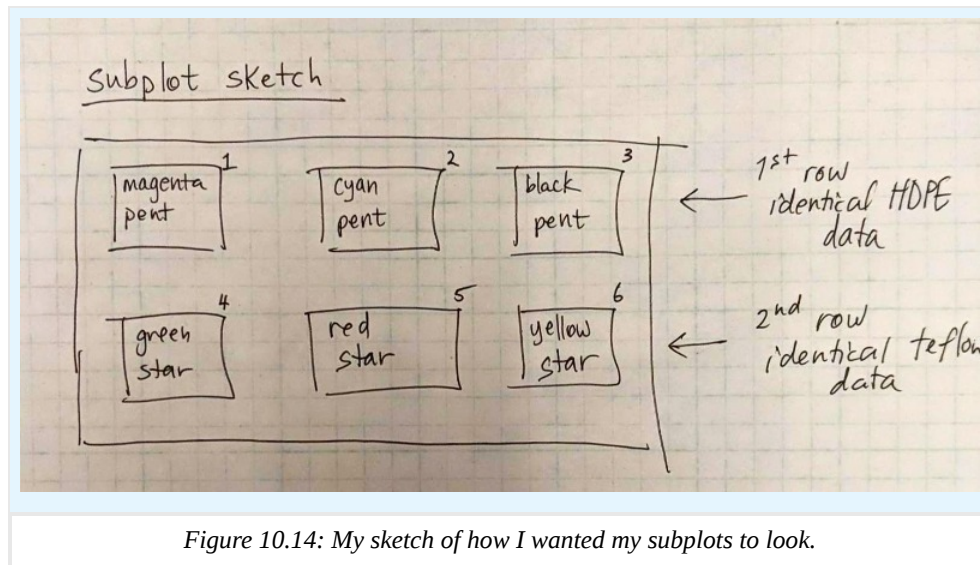


Figure 10.13: Example of 6 plots in the same figure using the `subplot()` command.

Try It!

Go ahead and try creating a 2x3 subplot grid with our tensile test data. You can reuse the data, so this IS a contrived example after all! But it is good practice.



I suggest sketching out your plan. Change the colors between each subplot and write them out on paper BEFORE you code to make sure you understand what is going to happen.

It is ok to struggle a little bit here! Don't jump straight to the solution until you try it! Subplots are kind of complicated and not straightforward but you need to practice.

1 % Insert your subplot code here:

run

run all

restart

restart & run all

Subplot Solution

To create the 6 subplots shown in Figure 10.14 above, this is what I added to my script.

```
1 %% Chapter 10 Follow Along Work
2
3 % Define the force vector
4
5 force = 0:5000:40000;
6
7 % Define the length vector
8
9 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
10
11 %Calculate area based off information
12
13 area = pi*0.005^2;
14
15 len_0 = 0.025; %initial length
16
17
18 %Calculate stress and strain of HDPE according to equations
19
20 e_stress = force / area;
21
22 e_strain = (len-len_0)/len_0;
23
24 %Calculate stress and strain of teflon
```

run

run all

restart

restart & run all

Figure 10.15: Solution to `subplot()` . Try it!

Titles, Labels, and Legends

The plots that we have created so far looked good, but they lack the information necessary to make them *good plots*. Namely, they need axes labels, titles, and legends. Don't worry! The hardest part of this chapter is done (looking at you SUBPLOT).

For now, comment out all of the stuff you have about subplots. We are going to concentrate on the plot with the HDPE and Teflon data on the same axes. If you have been following along, you should have something very similar to figure 10.16 below. We are going to *start* at line 34, just below all this stuff.

```

1 %% Chapter 9 Follow Along Work
2
3 clear
4 clc
5 close all
6
7 % Define the force vector
8 force = 0:5000:40000;
9
10 % Define the length vector
11 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
12 len_teflon = [25 28.2 31.4 34.5 37.8 40.8 44.1 47.3 50.5]/1000;
13
14 % Calculate area based off information
15 area = pi*0.005^2;
16
17 len_0 = 0.025; % initial length
18
19 % Calculate stress and strain according to equations
20 e_stress = force / area;
21 e_strain = (len-len_0)/len_0;
22
23 e_stress_teflon = force / area;
24 e_strain_teflon = (len_teflon-len_0)/len_0;
25
26 %plot(e_strain,e_stress) % original plot command
27 %plot(e_strain,e_stress,'g'); % plot a green line
28 %plot(e_strain,e_stress,'k<-.'); % plot black left arrows (dashdot line)
29
30 plot(e_strain,e_stress,'mp--'); % plot magenta pentagrams (no line)
31 hold on
32 plot(e_strain_teflon,e_stress_teflon,'g*--');
33 hold off

```

Figure 10.16: Example script of where you need to be. You need to have this and be ready to type on line 34.

When you hit run on the script above, you should get a figure with magenta pentagrams connected by a dotted line and green stars connected by a dotted line on the same axes. Note that the magenta stars are the HDPE data and the green stars are the Teflon data.

Adding a Title

To add a title to the plot, use the `title()` function. That is it. The title function accepts a character string input. So if we wanted to create a title for the plot called “Tensile Test Data” we would add a new line to our script that looks like this:

```
title('Tensile Test Data');
```

Go ahead and add that line to your script and re-run it. I told you it was easy! Just remember that the input to the `title()` function must be a character string which just means it needs `'` marks surrounding the text.

```

1 % Try it 10.6
2 % Add a title to the plot
3
4 % Define the force vector
5
6 force = 0:5000:40000;
7
8 % Define the length vector
9
10 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
11
12 %Calculate area based off information

```

```
13
14 area = pi*0.005^2;
15
16 len_0 = 0.025; %initial length
17
18
19 %Calculate stress and strain of HDPE according to equations
20
21 e_stress = force / area;
22
23 e_strain = (len-len_0)/len_0;
```

run

run all

restart

restart & run all

Adding Axes Labels

To add axes labels to the plot, we are going to use the `xlabel()` and `ylabel()` functions that correspond to the x-axis and y-axis respectively. Just like with the `title()` function we need to input the character string. Since in this particular plot the x-axis corresponds to the strain which is unitless and the y-axis corresponds to stress which has units of Pascals, we just need to add the following two lines of code:

```
xlabel('Strain (ratio)');
ylabel('Stress (Pa)');
```

Add those lines, re-run the script and watch the magic happen!

```
1 % Try it 10.7
2 % Add a x and y label to the plot
3
4 % Define the force vector
5
6 force = 0:5000:40000;
7
8 % Define the length vector
9
10 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
11
12 %Calculate area based off information
13
14 area = pi*0.005^2;
15
16 len_0 = 0.025; %initial length
17
18
19 %Calculate stress and strain of HDPE according to equations
20
21 e_stress = force / area;
22
23 e_strain = (len-len_0)/len_0;
24
```

run

run all

restart

restart & run all

Adding a Legend

MATLAB makes creating legends easy, but you **have to remember the order in which you plotted your data**. In this example, we plotted the HDPE data *before* the Teflon data. So to add an appropriate legend we would use the `legend()` command, and then supply character string inputs that correspond to the data. For example:

```
legend('HDPE', 'Teflon');
```

```

1 % Try it 10.8
2 % Add a legend to the plot
3
4 % Define the force vector
5
6 force = 0:5000:40000;
7
8 % Define the length vector
9
10 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
11
12 %Calculate area based off information
13
14 area = pi*0.005^2;
15
16 len_0 = 0.025; %inital length
17
18
19 %Calculate stress and strain of HDPE according to equations
20
21 e_stress = force / area;
22
23 e_strain = (len-len_0)/len_0;
24

```

run

run all

restart

restart & run all

Notice that when you add that line to your script and re-run it, you get a nice little legend! Nifty huh? There are some more advanced functionalities to the `legend()` function that won't be discussed in this book. But if you are interested type in `>> help legend` and check out some of the other features!



Figure 10.17: Sadly, this legend cannot be created in MATLAB.

The Final Script

If you have been following along, this is what your final script should look like. Keep in mind the subplot stuff is commented out below. If I were you, I wouldn't delete it!

```
1 %% Chaoter 10 Follow Along Work
2
3 % Define the force vector
4
5 force = 0:5000:40000;
6
7 % Define the length vector
8
9 len = [25,27,29,31.1,32.9,35,37,39,40.8]/1000;
10
11 %Calculate area based off information
12
13 area = pi*0.005^2;
14
15 len_0 = 0.025; %initail length
16
17
18 %Calculate stress and strain of HDPE according to equations
19
20 e_stress = force / area;
21
22 e_strain = (len-len_0)/len_0;
23
24 %Calculate stress and strain of teflon
```

run

run all

restart

restart & run all

Figure 10.18: My script up until this point.

Plotting Functions

Once you have mastered plotting in MATLAB plotting functions is actually very easy. For example, lets make a simple plot of $y = x^3 - x^2$ from $x = -0.5$ to $x = 2$.

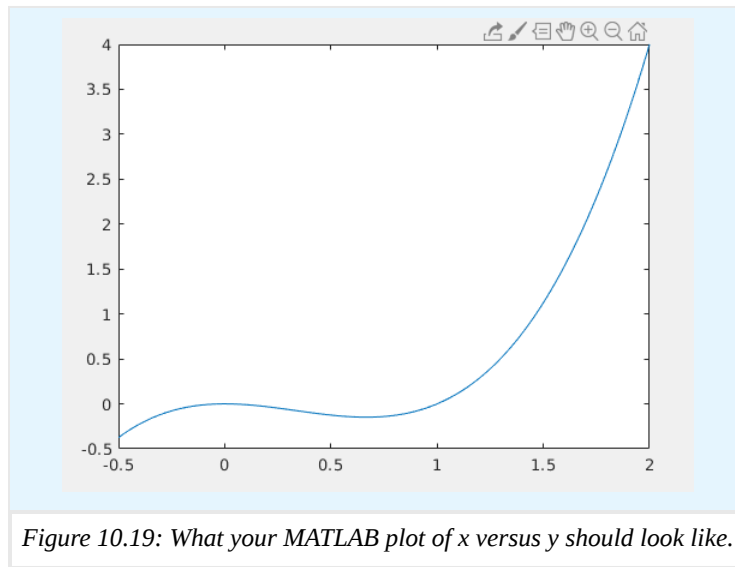
In the previous chapter, there was an example that showed us how to evaluate a function at different values. The key was to first, create an array of (x) values, then mathematically evaluate those (x) values with the mathematical function of interest. We will do the same thing here. In your script, type:

```
>> x = -0.5:0.01:2;
>> y = x.^3 - x.^2;
```

Notes: We choose an increment of 0.01 so that the graph looks “smooth”. What we are really doing is creating a non-smooth graph but since the increment is so small, it appears smooth. Also, remember, we need the `.^` because we are performing exponentiation on the (x) array.

Now, lets plot (x) versus (y).

You should see a graph pop up that looks like figure 10.19. That's all there is to it!



This page titled [4.4: MATLAB Plotting](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 10: MATLAB Plotting](#) is licensed [CC BY-NC 4.0](#).

4.5: User Defined Functions



Figure 15.1: Luckily for us, MATLAB functions aren't THIS complicated.

Functions are a fundamental part of mathematics and most students are familiar with how they work. In math class, you were likely taught that you can think of functions like little machines (figure 15.2) that associate a unique number y , with any given number, x . You may have even learned the terminology that the x value is the input, and the y value is the output.

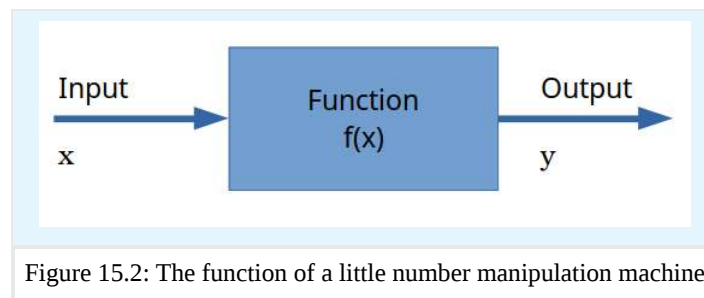


Figure 15.2: The function of a little number manipulation machine

If you can understand how a mathematical function works, you are actually well on your way to understanding how MATLAB functions work! MATLAB functions work in the same way as mathematical functions. You have one (or more inputs), manipulate the input(s) according to some rules, and then you return one (or more outputs).

Learning Goals

In this chapter, we will learn the how to create our own MATLAB functions and why we should. That means learning:

- What MATLAB functions are and why they are important.
- Reviewing some of the most common MATLAB functions we have already been using.
- Where the help text of a function lives and why it is important to write good help text.

MATLAB Built-In Functions

As mentioned above, a MATLAB function is very similar to a mathematical function. You have already been using MATLAB functions throughout this book! I have been careful to distinguish functions from other bits of code by appending a set of parenthesis on the end whenever we are discussing functions (i.e. `sin()`). Let's investigate a few MATLAB built-in functions to understand how functions work:

- `sin()`
- `mean()`
- `max()`

Try It!

It is worth taking a moment and reading the help text for the 3 functions listed above. To do so, open MATLAB and type in `>> help function_name` into the command window. You should replace `function_name` with the function of interest (e.g. `>> help max`). You do not need to memorize anything but go ahead and read through the help texts to get an idea of how they work.

```
1 % Remember, Octave has help text too!  
2  
3 help
```

runrun allrestartrestart & run all

`sin()` Function

The `sin()` function is perhaps the easiest to understand because you are likely already familiar with trigonometric functions. The `sin()` function takes an **input**, angle in radians, and returns an **output**, the ratio of the length of the side of a right triangle that is opposite that angle to the length of the longest side of the triangle.

An example of how to use the `sin()` function would look like this (make sure you are trying it for yourself in MATLAB!):

```
1 x = 2.1; % in radians  
2  
3 ratio = sin(x)
```

run

run all

restart

restart & run all

Notice how the **output** of the `sin()` function is assigned to the `ratio` variable.

Now the real question is, how does MATLAB calculate that ratio? Well, MATLAB engineers have created thousands of numerical algorithms stored in **functions** that they supply with the product. This brings us to the central idea behind functions: functions exist to make it easy to run snippets of code with different inputs.

Key Concept: Functions exist to make it easy to run snippets of code with different inputs.

An example would be the sine trigonometric function. Most engineers and scientists do not want to have to program the algorithm that calculates the sine of an angle every time they need to calculate sine. That is too much work! So you can create a function that calculates the sine of an angle, name it something that makes sense (in this case `sin()`), and allow it to work for any input!

`mean()` Function

Recall that the `mean()` function accepts an array as in **input**, and returns the average value of the values inside the array as the **output**.

An example of how to use the `mean()` function could look like this:

```
1 array = [9 3 1 3 5];
2
3 avg = mean(array)
```

run

run all

restart

restart & run all

When you try this example, you should notice that the value stored in the variable `average` is 4.8, the mean of the array! Keep in mind that you could have used a few different functions to calculate the exact same value. Recall: $\{\text{average}\} = \{\text{sum}\} / \{\text{count}\}$. Therefore:

```
1 array = [9 3 1 3 5];
2
3 % avg = mean(array)
4
5 avg_long_way = sum(array)/length(array)
```

run

run all

restart

restart & run all

Again give it a shot. You will notice that the value in `avg` is the same as `avg_long_way` which is unsurprising. However, you can imagine that if you had to constantly type out the long way, it would get tedious. The other thing to consider is that `sum()` and `length()` are functions themselves! Take those away and this problem of calculating a mean gets really complicated.

Key Concept: Functions make your code easier to read and easier to type.

The key concept here is that functions make your code easier to read and easier to type. So when you have an algorithm that is likely to get run over and over again, it is a good idea to make a function for it!

`max()` Function

Next up, let's consider the built-in function `max()`. This particular function is interesting because of its flexibility. The function accepts an array as an **input** and returns the maximum value in that array as its **output**. That in itself is cool, but not that exciting (hopefully you read the help text for `max()` and know where this is going). The exciting part is that `max()` can *optionally* return the index where the maximum value is located within the array!

To see this in action, let's try an example. We can use the array variable still located in our workspace from the previous example:

```
1 array = [9 3 1 3 5];
```

```
2  
3 biggest = max(array)
```

run run all restart restart & run all

That will store the number 9 inside the `biggest` variable. Cool, but not exciting. Take an account of your workspace variables. Now try:

```
1 array = [9 3 1 3 5];  
2  
3 [maximum, index] = max(array)
```

run run all restart restart & run all

Look at your workspace now. Notice that it created two new variables, one called `maximum` that has the number 9 stored in. The execution of that code also created a new variable, `index` with the number 1 stored in it. The 9 is the biggest number in the array, and the 1 is the index of where the number is located.

The first key concept to learn from this example is that MATLAB functions can accept multiple **inputs** and can return any number of **outputs**.

Key Concept: Functions can accept multiple inputs and can return any number of outputs.

To be specific, when you are creating a function:

- You can *require* any number of **inputs**. For example, you can require your user to specify at least one array, or you can say that a minimum of two arrays is required, etc.
- You can make any number of **inputs** optional. For example, you can have one required, and one optional input.
- The only output that is *not optional* is the first output. **All other outputs from a function are optional.** That means that you can not force your user to accept two outputs from your function. This is an important thing to consider when designing your own functions.

Let's continue with the `max()` function to investigate our next key concept. If you have been doing your brain workouts, you read through the `max()` help text. See figure 15.3 below, specifically the highlighted region.

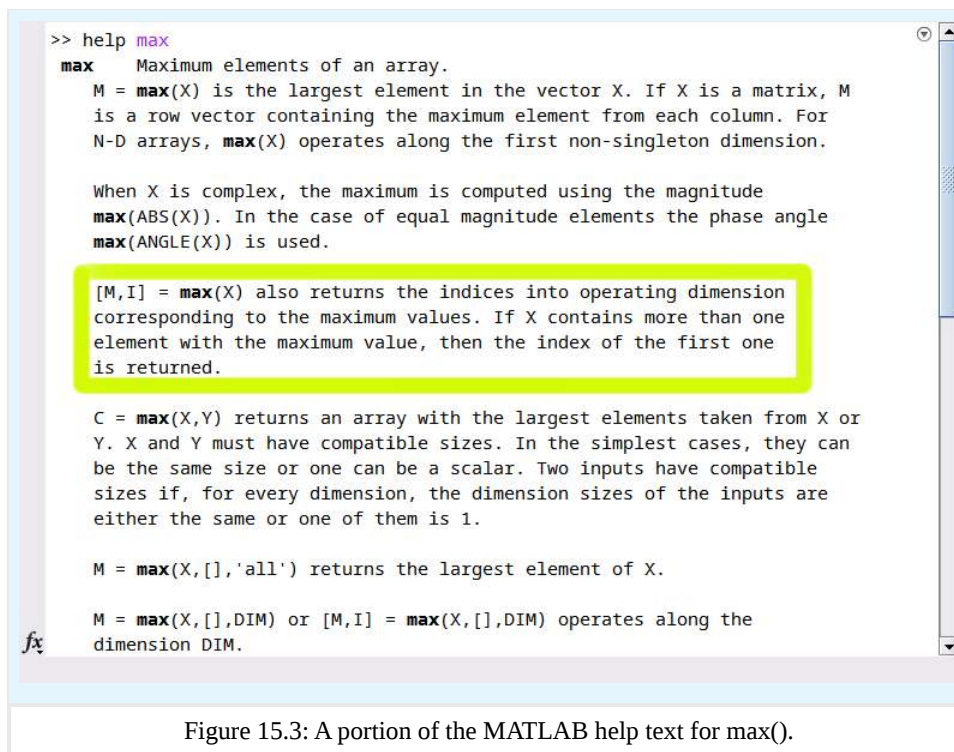


Figure 15.3: A portion of the MATLAB help text for `max()`.

Notice how well this function is described! It shows an example of how to use the function (`[M,I] = max(X)`) and gives a nice description of how it works.

The second key concept is that well-written help text is critical to writing a good function. If the documentation for this function was poorly written or confusing, no one would use it!

Key Concept: Well-written help text is critical to writing a good function.

A good function is one that is well documented, not only for your users but for yourself! Variable names and algorithms always make sense when you are working on them, but when you put them down for a while, you need good documentation to ensure that your code is still usable.

User Defined Functions

Now that we have seen how MATLAB built-in functions work and what they are used for, it is time to create our own functions. However, before we can begin making our own functions, we need to have a quick review of the current working directory.

The Importance of the Current Working Directory

Up until this point, you could have gotten away with bad file management. You could store all of your files in the same folder, and put them all on your Desktop, it hasn't mattered. However, if you want to use your function files, you are going to have to be cognizant of your current working directory and your path.

As an analogy, think of MATLAB as a mechanics garage and think of functions as different tools. MATLAB comes pre-loaded with a bunch of tools (`sin()` , `length()` , `max()` , etc) and already knows where they are located. When you make a new tool called `impact_driver()` , you can store it anywhere. Maybe you are storing your new `impact_driver()` in the shed out back (your My Documents folder). Maybe you threw it into the lake. MATLAB has no idea where to look for it. So when you tell MATLAB you want to use your new `impact_driver()` it does the logical thing, it looks where you are standing (your current working directory). So if you tell MATLAB, "I want to use my new `impact_driver()` ", and you are standing in your shed, MATLAB will check the shed for the tool. If it can't find it in the shed, MATLAB will check the garage (this is called the path). If it can't find it where you currently are (your current working directory) or the garage (the path) then it gives up. I don't blame it! It isn't a magician.

The moral of the story is that MATLAB will *always look in the current working directory for your user-defined functions*.

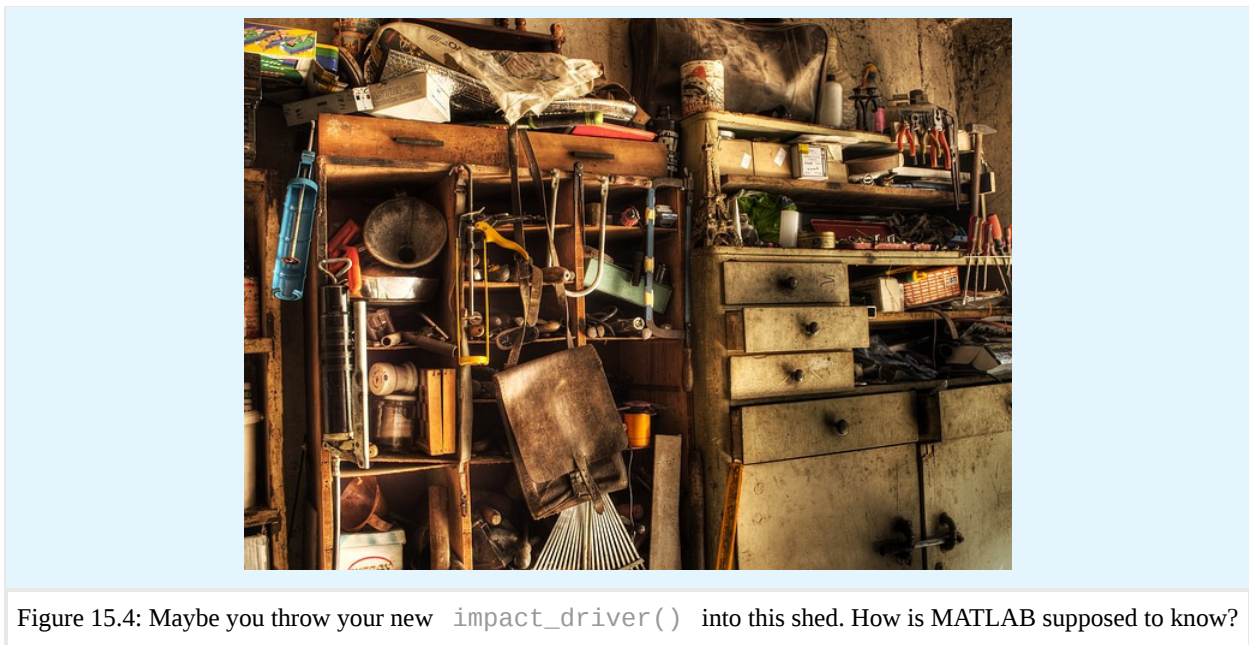


Figure 15.4: Maybe you throw your new `impact_driver()` into this shed. How is MATLAB supposed to know?

This is important to remember when you save a function and are trying to run it. If the function is not located in either the path (the garage in our analogy) or the current working directory, it will throw an error.

`sphere_vol()` - Your First Function

Before we start creating functions remember our mantra! **Think, sketch, code, test, repeat!** This will help us organize our thoughts and make sure that we are making the function correctly.

Imagine that you are working on a program that requires you to calculate the volume of a sphere several times. Recall: $volume\ of\ sphere = \frac{4}{3}\pi r^3$ so you are constantly writing that formula over and over again.

```
>> volume1 = 4/3*pi*2^3;  
  
>> volume2 = 4/3*pi*3^3;  
  
...
```

This is a perfect candidate for a function! We can create a function that automatically calculates the volume and returns it. This will simplify our code! Now it will look like this:

```
>> volume1 = sphere_vol(2)  
  
>> volume2 = sphere_vol(3)  
  
...
```

Notice how it is much easier to read and understand. In the lower example, it is clear that the input is 2 for `volume1` and 3 for `volume2`. Not only that, but it is easier to type and the user is less likely to have errors.

Think - What will the input variables be? Will your user expect your function to work with an array of radii? What about the output variables? What will the user expect to get back? The mathematics in this function isn't super complicated but make sure that you understand *how* to program MATLAB to correctly calculate the volume from an array and scalar input.

How to Create a Function

Now that you understand the importance of the current working directory, navigate to a folder (or better yet create a new folder) to store your new function file. We are going to create a function called `sphere_vol()` that will take radius, `r`, as an **input**, and return the volume, `V` as an **output**.

To start, click the “New” drop-down arrow on the “Home” tab of the MATLAB GUI toolbar. Click the “Function” option (see figure 15.5 below).

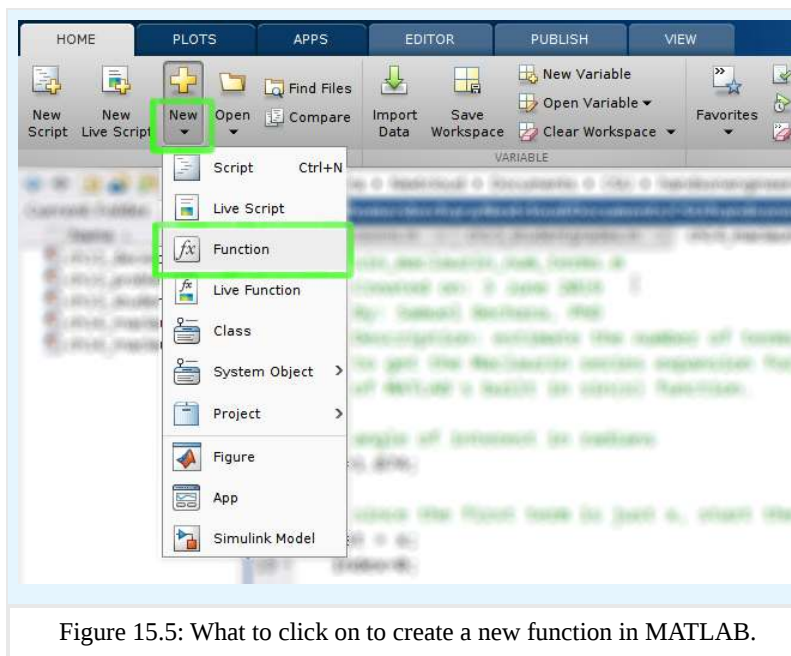


Figure 15.5: What to click on to create a new function in MATLAB.

Once click the correct button, a new editor window should pop up that looks like figure 15.6 below. MATLAB creates a template function that we can then modify to fit our needs. The reality is there is no difference between a script file and a function *except that a function must begin with the keyword `function`* .

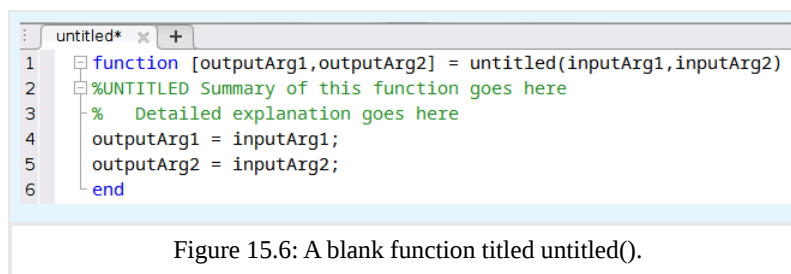


Figure 15.6: A blank function titled untitled().

Take a second to look at this template function. Let's breakdown the very first line of code in this template function:

```
function [outputArg1, outputArg2] = untitled(inputArg1, inputArg2)
```

- The very first thing to notice is that the function template *is structured in the exact same way as you would use a function*. For example, when we talked about the `max()` function above we did: `>> [maximum, index] = max(array);` The only difference is that the function definition starts with the word `function` . Makes it easy to remember!
- Notice how the *very first word on the very first line is `function`* . This must be true for MATLAB to understand this file as a function. You can not put *anything* above that word. Even a comment `%` preceding the word `function` will fool MATLAB.
- After `function` you can see the two **output** variables, `[outputArg1, outputArg2]` . You list your output arguments in square brackets `[]` separated by a comma.

- Next, we have an `=` sign. Again, this mimics how you would use a function.
- The function name comes after the `=` sign and precedes the parenthesis. In the default case, the function name is `untitled()`. I will mention this again, but the function name needs to be *exactly the same* as the filename.
- Notice the **input** variables (`inputArg1`, `inputArg2`) are listed inside `()` and separated by a comma.
- Finally, there is no `;` at the end of a function line! This is a common mistake introductory programmers make.

Try It!

Brain workout time. This is a great opportunity to push your mental muscles a bit.

Sketch - Remember our goal is to create a function that can calculate the volume of a sphere. Before you start changing things in MATLAB, grab a pen and a piece of paper and write out what you *think* the first line should be. Recall that our function name is going to be `sphere_vol()`. We are going to have an **input** variable that is a scalar radius or array of radii. We will have an **output** variable that is a scalar volume or an array of volumes.

Did you give it your best shot? This is what I came up with.

```
function [volume] = sphere_vol(radii)
```

Does yours look similar? Keep in mind that the variables `volume` and `radii` are what I *chose*. They could have been *anything*. You could still be 100% correct if you picked different variable names!

At this point go ahead and edit the function template according to my first line above (or even better, use your own variables) and then save your function. Click the save button in the MATLAB GUI. When the save dialogue box opens up, notice how it automatically fills in the function name in the “File Name” text edit box (figure 15.7 below). **DO NOT CHANGE THE FILE NAME!** If you change it, your function will not work. Don’t forget about the current working directory!

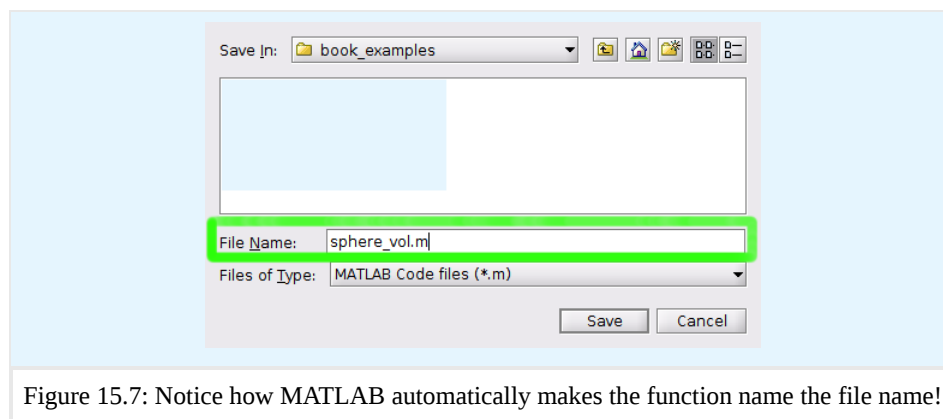


Figure 15.7: Notice how MATLAB automatically makes the function name the file name!

The last thing to notice is your “Current Folder” browser. Notice the icon looks different than a script file! It has a little f_x in the icon instead of the MATLAB logo. This is another little visual indicator when you are browsing your folders that it isn’t a script file, but is a function file.

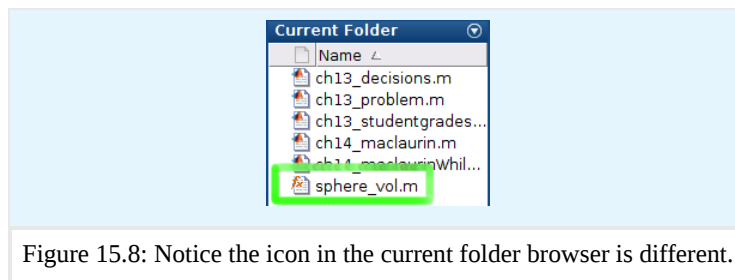


Figure 15.8: Notice the icon in the current folder browser is different.

Creating Helpful help Text

Remember that helpful text that popped up when we typed in `>>help max` into the command window? We need to create that for our functions too. In our **think** and **sketch** phases, we have come up with an idea for how this program should work. Now let's create help text so that when someone types in `>>help sphere_vol` into the command window, they get helpful hints for how to use our function.

When someone types in `>>help function_name` MATLAB returns all commented lines `%` below the initial function definition line. In the case of our template function, that is:

```
%UNTITLED Summary of this function goes here
```

```
% Detailed explanation goes here
```

You can see this in action! Go ahead and type `>>help sphere_vol` in the command window. See the text? Well, that isn't very helpful. So let's replace it so that it is actually helpful. See figure 15.9 below for how I defined my help text.

```
1 function [volume] = sphere_vol(radii)
2
3 % sphere_vol calculates the volume of a sphere
4 % This function accepts either a scalar radius or array of radii and returns a scalar or array of vol
5 % Inputs:
6 % radii - a scalar radius or array of radii
7 % Outputs:
8 % volume - a scalar volume or array of volumes
9
10 outputArg1 = inputArg1;
11 outputArg2 = inputArg2;
12
13 end
```

run run all restart restart & run all

Figure 15.9: Our sphere_vol() function up to this point.

There are a couple of good practices to notice. You can technically put whatever you want for your help text. However, good practices state that you should include:

- The first line should be the function name, followed by a short description of the function.
- The next line(s) should then include a more detailed description of the function. Its purpose, and how it works. In this case, our function is pretty straightforward but as you make more functions, they will likely get more complicated and require more descriptions.
- After the description, include a list of all **input** and **output** variables and what they should be.

When you type this help text in, save the file, and then try typing `>>help sphere_vol` into the command window again. Notice how you now get your useful help text! Neat huh?

```
1 % Try running this general function to see what the help text looks like for Octave. Feel free to edit
2
3 function [output] = func_name(input)
4
5 % This is a test script to show textbook readers how help text works for user defined function in Octav
6 % This function only returns the user's input back to them
7
8 output = input
9
10 end
11
12 help func_name
```

runrun allrestartrestart & run all

The Meat (i.e. Math) of the Function

Now that we have everything defined and sketched out, it is time to program the function's functionality! To do so let's erase the two `outputArg...` lines that came with the template and program in our function. Make sure not to delete the last line, the `end` keyword! All functions must start with the word `function`, and end with a line that just has `end` on it.

Code - Since this is just a simple function, we can accomplish this in one line of code:

```
volume = (4/3)*pi*(radii.^3);
```

That is it! Notice that we had to use the `.^` to account for the possibility that our user would input an array of radii. If the user just inputs a scalar value, the function will still work. At this point, we have completed the function. It should look similar (if you used your own variable names, those will be slightly different) to figure 15.10 below.

```
1 function [volume] = sphere_vol(radii)
```

```
2
3 % sphere_vol calculates the volume of a sphere
4 %   This function accepts either a scalar radius or array of radii and returns a scalar or array of vol
5 % Inputs:
6 %   radii - a scalar radius or array of radii
7 % Outputs:
8 %   volume - a scalar volume or array of volumes
9
10 volume = (4/3)*pi*(radii.^3)
11
12 end
```

run

run all

restart

restart & run all

Figure 15.10: Our completed sphere_vol() function!

Function Variables vs User Variables

Before I show you how to use your function, it is important to note the distinction between the internal function variables and the variables that your user uses as `inputs` and `outputs`.

Key Concept: The user can use different variable names than the ones that you internally defined for your function.

This is a very important point and one that I see undergraduate programmers constantly confused about. To show you what I mean, let's use your function in the command window (or you can create a script if you would like).

First, let's define an array of radii (make sure you are following along with this example!):

```
>> r = 1:10;
```

Next, let's use our new function `sphere_vol()` to calculate the volumes associated with those radii! Exciting!

```
1 function [volume] = sphere_vol(radii)
2
3 % sphere_vol calculates the volume of a sphere
4 %   This function accepts either a scalar radius or array of radii and returns a scalar or array of vol
5 % Inputs:
6 %   radii - a scalar radius or array of radii
7 % Outputs:
8 %   volume - a scalar volume or array of volumes
9
10 volume = (4/3)*pi*(radii.^3);
11
12 end
13
14 r = 1:10;
15
16 v = sphere_vol(r)
```



Remember that our function's internal variables were `radii` and `volume`. *You do not need to use these exact variable names when using your function.* In this case, we used the variables `r` for the radii and we stored the volumes in variable `v`. The internal function variables are just placeholders that tell the function how to manipulate the **inputs** and where to store the **outputs**. The names inside the function do not matter to the user.

Test - Your function is ready to use! Go ahead and test the function with a bunch of different radii. Can you break the function? Make it throw an error?

Repeat - Again, this is a pretty simple function, but when you make more complicated functions it is a good idea to revise your help text or your sketches as you test and find issues.

Adding a Folder to the Path

If you are the organized type, you can actually add folders to your MATLAB path. Returning to our garage analogy: you can think about it as throwing the `impact_driver()` function into a box, and throwing it in the garage (path).

Adding folders to the path is a good way to increase the usefulness of your functions. Right now, the `sphere_vol()` function we created *will only work if you are currently in the directory that contains the file sphere_vol.m*! If we add the folder that `sphere_vol.m` is contained into the path, we can call the `sphere_vol()` function from anywhere.

The idea is that once you create this cool new function, you might want to use it in scripts that are stored in different folders. You can only do this if you add the function file location to MATLAB's path.

To show you how this works, let's create a folder called "Functions" and store it somewhere on your computer that you will remember (MyDocuments? Somewhere else? You decide). Then navigate to the folder that contains this new folder in MATLAB. In the "Current Folder" browser you can then right-click on the "Functions" folder, and select the "Add to path >" option. Then select the "Selected Folder and Subfolders" option. Viola! Now any function file inside this particular folder will be accessible regardless of your current working directory. You told MATLAB where to look!

This page titled [4.5: User Defined Functions](#) is shared under a [CC BY-ND 4.0](#) license and was authored, remixed, and/or curated by [Julio Terán](#).

- [Chapter 15: User Defined Functions](#) is licensed [CC BY-NC 4.0](#).

Index

A

Amazon

1.9: References

arithmetic operators

3.4: Programming in any language

artificial intelligence

1.3: Data Science in Practice

B

big data

1.2: What Is Data Science?

1.3: Data Science in Practice

1.4: Data and Datasets

bivariate

1.6: Correlation and Linear Regression Analysis

business analytics

1.3: Data Science in Practice

C

C++

3.4: Programming in any language

calculator versus programming language

3.4: Programming in any language

cloud storage

1.8: Chapter Review

comparison of languages

3.4: Programming in any language

computer operators

3.4: Programming in any language

concepts

1.1: Introduction

conditional statements

3.4: Programming in any language

control statements

3.4: Programming in any language

correlationanalysis

1.6: Correlation and Linear Regression Analysis

correlationcoefficients

1.6: Correlation and Linear Regression Analysis

customer habits

1.9: References

D

data

1.7: Key Terms

data analysis

1.1: Introduction

1.2: What Is Data Science?

1.4: Data and Datasets

1.7: Key Terms

1.8: Chapter Review

data analytics

1.9: References

data collection

1.4: Data and Datasets

1.7: Key Terms

1.8: Chapter Review

data formats

1.4: Data and Datasets

data management

1.2: What Is Data Science?

1.8: Chapter Review

data preparation

1.7: Key Terms

1.8: Chapter Review

data processing

1.2: What Is Data Science?

1.8: Chapter Review

data reporting

1.2: What Is Data Science?

1.8: Chapter Review

data science

1.1: Introduction

1.2: What Is Data Science?

1.3: Data Science in Practice

1.4: Data and Datasets

1.7: Key Terms

1.8: Chapter Review

1.9: References

data structures

1.4: Data and Datasets

data visualization

1.2: What Is Data Science?

1.7: Key Terms

data warehousing

1.8: Chapter Review

dataanalysis

1.5: Using Technology for Data Science

datascience

1.6: Correlation and Linear Regression Analysis

dataset

1.4: Data and Datasets

1.7: Key Terms

DataVisualization

1.5: Using Technology for Data Science

decision making

1.2: What Is Data Science?

documentation (programming)

3.3: Documentation and such

E

education

1.3: Data Science in Practice

Excel

1.5: Using Technology for Data Science

1.7: Key Terms

extreme documentation

3.3: Documentation and such

F

farming efficiency

1.9: References

flowcharts

3.3: Documentation and such

Fortran

3.4: Programming in any language

fraud detection

1.9: References

functions

3.4: Programming in any language

G

GDL/IDL

3.4: Programming in any language

H

health care

1.3: Data Science in Practice

I

IDL/GDL/PVWave

3.4: Programming in any language

if statements

3.4: Programming in any language

input and output statements

3.4: Programming in any language

insights

1.4: Data and Datasets

interdisciplinary

1.8: Chapter Review

Internet of Things

1.3: Data Science in Practice

IoT

1.9: References

L

learningobjectives

1.6: Correlation and Linear Regression Analysis

linearregression

1.6: Correlation and Linear Regression Analysis

logic operators

3.4: Programming in any language

lookfor (MATLAB/Octave)

3.3: Documentation and such

loops

3.4: Programming in any language

M

machine learning

1.2: What Is Data Science?

1.3: Data Science in Practice

machinelearning

1.5: Using Technology for Data Science

MATLAB/Octave/Scilab

3.4: Programming in any language

O

Octave/MATLAB

3.4: Programming in any language

P

pandas

1.7: Key Terms

petroglyphs

1.1: Introduction

PowerBI

1.5: Using Technology for Data Science

predictive analytics

1.3: Data Science in Practice

programming languages

1.1: Introduction

programminglanguages

1.5: Using Technology for Data Science

pseudocode

3.3: Documentation and such

python

- [1.1: Introduction](#)
- [1.5: Using Technology for Data Science](#)
- [1.6: Correlation and Linear Regression Analysis](#)
- [1.7: Key Terms](#)
- [3.4: Programming in any language](#)

Q

qualitative data

- [1.4: Data and Datasets](#)

quantitative data

- [1.4: Data and Datasets](#)

R

R

- [1.5: Using Technology for Data Science](#)

read and write

- [3.4: Programming in any language](#)

regressionanalysis

- [1.6: Correlation and Linear Regression Analysis](#)

Relational Operators

- [3.4: Programming in any language](#)

S

scatterplots

- [1.6: Correlation and Linear Regression Analysis](#)

scripts

- [3.4: Programming in any language](#)

sensors

- [1.9: References](#)

single example using different computer

languages

- [3.4: Programming in any language](#)

smart cities

- [1.3: Data Science in Practice](#)
- [1.9: References](#)

spreadsheets

- [1.1: Introduction](#)

statistical analysis

- [1.2: What Is Data Science?](#)

statistical software

- [1.1: Introduction](#)

statisticalrelationship

- [1.6: Correlation and Linear Regression Analysis](#)

statisticalsoftware

- [1.5: Using Technology for Data Science](#)

subroutines

- [3.4: Programming in any language](#)

T

Tableau

- [1.5: Using Technology for Data Science](#)

technology

- [1.1: Introduction](#)

terminology

- [1.1: Introduction](#)

V

variables and arrays

- [3.4: Programming in any language](#)

W

Walmart

- [1.9: References](#)

write and read

- [3.4: Programming in any language](#)

writing good programs

- [3.3: Documentation and such](#)

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: [Data and Programming - An Overview](#)

Webpages: 34

Applicable Restrictions: No Derivatives

All licenses found:

- [CC BY-ND 4.0](#): 91.2% (31 pages)
- [Undeclared](#): 5.9% (2 pages)
- [CC BY 4.0](#): 2.9% (1 page)

By Page

- [Data and Programming - An Overview](#) - [CC BY-ND 4.0](#)
 - [Front Matter](#) - [CC BY-ND 4.0](#)
 - [TitlePage](#) - [CC BY-ND 4.0](#)
 - [InfoPage](#) - [CC BY-ND 4.0](#)
 - [Table of Contents](#) - [Undeclared](#)
 - [Licensing](#) - [CC BY-ND 4.0](#)
 - [1: What Are Data and Data Science?](#) - [CC BY-ND 4.0](#)
 - [1.1: Introduction](#) - [CC BY-ND 4.0](#)
 - [1.2: What Is Data Science?](#) - [CC BY-ND 4.0](#)
 - [1.3: Data Science in Practice](#) - [CC BY-ND 4.0](#)
 - [1.4: Data and Datasets](#) - [CC BY-ND 4.0](#)
 - [1.5: Using Technology for Data Science](#) - [CC BY-ND 4.0](#)
 - [1.6: Correlation and Linear Regression Analysis](#) - [CC BY 4.0](#)
 - [1.7: Key Terms](#) - [CC BY-ND 4.0](#)
 - [1.8: Chapter Review](#) - [CC BY-ND 4.0](#)
 - [1.9: References](#) - [CC BY-ND 4.0](#)
 - [2: Units, Dimensions, and Conversions](#) - [CC BY-ND 4.0](#)
 - [3: Introduction to programming and logic](#) - [CC BY-ND 4.0](#)
 - [3.1: Thinking Algorithmically](#) - [CC BY-ND 4.0](#)
 - [3.2: Relational and Logical Operators](#) - [CC BY-ND 4.0](#)
 - [3.3: Documentation and such](#) - [CC BY-ND 4.0](#)
 - [3.4: Programming in any language](#) - [CC BY-ND 4.0](#)
 - [3.5: Mathematical Models](#) - [CC BY-ND 4.0](#)
 - [4: MATLAB](#) - [CC BY-ND 4.0](#)
 - [4.1: Introduction to MATLAB](#) - [CC BY-ND 4.0](#)
 - [4.2: MATLAB Scripting and Arrays](#) - [CC BY-ND 4.0](#)
 - [4.2.1: MATLAB Decisions](#) - [CC BY-ND 4.0](#)
 - [4.3: MATLAB Loops](#) - [CC BY-ND 4.0](#)
 - [4.4: MATLAB Plotting](#) - [CC BY-ND 4.0](#)
 - [4.5: User Defined Functions](#) - [CC BY-ND 4.0](#)
 - [Back Matter](#) - [CC BY-ND 4.0](#)
 - [Index](#) - [CC BY-ND 4.0](#)
 - [Glossary](#) - [CC BY-ND 4.0](#)
 - [Detailed Licensing](#) - [Undeclared](#)