Discover Computing

An engineer's introduction to programming with MatLab

Brian Adams, Ph.D., PE

March 11, 2024

Pennsylvania State University

Disclaimer
You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes. dsdvs

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

Publisher
First printed in August 2020 by Pennsylvania State University

Contents

List of Figures

List of Tables

List of Listings

About this book

There are many books on MatLab, and even more on programming and on computing. But few - if any - take a procedural approach to the subject. This text attempts to do just that.

Programming texts tend to follow a traditional approach. You learn how to print *Hello World*. From there you learn to calculate, followed by selection, repetition, arrays, and then finally functions. Some books may include recursion, but if they do it is near the end if not the last topic in the text.

MatLab texts consistently take a different approach. Since MatLab is a matrix centered language - after all, MatLab is short for *Matrix Laboratory* - it starts with matrices and how to use them to solve engineering problems. They tend to avoid the more traditional topics in programming until later, and then only as to how they support matrix operations.

This approach is paradoxical. While the student learns to use MatLab they might not receive the exposure to the traditional programming topics that they would in a course taught in Python, or C++. This is fine if the goal of the course is to prepare them to apply MatLab to specific problems in future engineering classes. But engineering students eventually graduate and move on to industry or graduate school. Again, not a problem if, as an engineer, they never need to code in a language other than MatLab. But the chance of that in today's technological world is slim.

This book bridges that gap between the traditional programming course and the MatLab course. Its goal is to introduce programming to an audience of engineers or engineering students with no prior experience in the topic. This approach starts with the traditional concepts of single value variables - or in the MatLab case, scalars - and then proceeds with the common programming structures.

Programming pedagogy has always been variables, selection, repetition, functions or procedures, and - if time permits - classes and objects. The problem with this approach is that while we start calling functions from the beginning - trig functions, logarithms, square roots - we do not learn about writing functions till near the end of the course. But functions are what make programming plug and play. Once there is a basic understanding of functions the student can start reusing code in multiple programs. It quickly becomes obvious that much of programming is repetition of what they have done before. To take advantage of this, procedural programming is not put off till the end, but in this approach it is the second topic of the text coming right after variables and assignments.

Once the foundation of functions is laid the course moves onto selection, and since they already have exposure to functions, recursion - another change from tradition. Why recursion and why before repetition? Recursion is rarely taught in beginning program classes. But recursion is natural. It is how the world actually operates. With the basics of writing functions already done it is a natural progression to recursion.

From recursion the book proceeds to repetition structures - while loops - and, with the for loop, the first look at lists, vectors, arrays, and matrices. This leads into to the traditional MatLab topics vectors, matrices, and solving simultaneous equations.

In a normal semester the first chapter is assigned to the students as a reading to provide some history, and programming actually starts in chapter 2. The goal of this text is not just learning to program but understanding computation and computing. This is the emphasis of the second chapter. It discusses computing, algorithms, flow charting, and algorithmic complexity - another topic rarely mentioned in an introductory course. Complexity provides a look at reality. While an algorithm can be developed and a program written, can the program actually run to completion in an amount of time so that a usable solution can be found?

Flow charts are another topic that while presented in the past has been dropped in most modern programming texts. In this book we bring it back. They provide an excellent visual representation of many of the concepts, and are thus used throughout the book.

My thanks go to all of the students who have already used this book, providing suggestions and identifying errors. I am hopeful that those who use it in the future will continue to do so with that same critical eye.

Joseph Brian Adams, Ph.D., PE
11 May 2021

Introduction to Computing and Computer Science

1

Computers have been around for thousands of years. They just did not have all of the cool flashing lights.

Computer Science encompasses nearly every facet of our lives. But as engineers our vocational interest in computer science tends to be more narrow. While we may be interested in designing a more powerful computer, or a more efficient computer, or a more practical computer, it is more likely that we are interested in the computer as a tool; a means of completing a computational task in the most efficient means possible. In this case our interest is not in the computer, but in computing.

1.1 Computing

Computing
Computing is the process of turning data into information.

Any elementary school students can provide you with a definition of computing; it is adding and subtracting. The high school student would tell you that it is using a computer. And the engineer is likely to return to the arithmetic definition. While these are all true, their definitions are too narrow. We are going to use a more general definition of computing. Computing is the process of turning data into information.

With a definition of computing we can now define a computer. A computer is a tool for computing. A circular definition but it describes the process of using a tool to turn data into information.

This definition is important in part for what it omits. It does not provide any indication or specifics. The computer does not have to be digital or even mechanical. It can be static or dynamic. As long as we can use the tool to turn data into information it is a computer. With this definition of computing we can see that computation has been around for thousands of years.

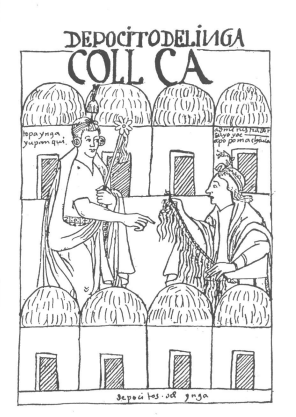


Figure 1.1: The Aztec Quipu was a system of strings and knots that were used to store numerical information.

It should be obvious that the first computers were ourselves. We counted on our fingers and toes. According to archaeologists the idea of counting goes back over fifty

thousand yearsIf the counts exceeded ten or twenty the ancients learned to make marks or cut notches in sticks.

The term *computer* actually comes from this. Before mechanical computers, when large scale computation was needed people were hired to do the calculations by hand.

This began in the nineteenth century when the Harvard Observatory needed people to perform tedious arithmetic calculations. Those hired were primarily women who were degreed mathematicians. This continued into the twentieth century and had become famous with its depiction at NASA in the 2016 film *Hidden Figures*. The job title of these women was *computer* and the term stuck.

And after we counted if we needed to store the resulting information we put pebbles in a pot or made knots in a string (figure 1.1). Later, we created ledgers and made notations in those ledgers. These were simple mechanical means of performing calculations and storing the results.

1.2 Measurement

Cubit
The cubit was an ancient unit of length that was based roughly on the length of the forearm from the extended middle finger to the elbow. While considered a standard its length varied between cultures.

As humanity progressed so did our demands for accuracy and precision. These demands drove innovation in computing. An early innovation came about as a result of mankind's desire for accurate and repeatable measurements.

Early construction was accomplished using measurements based upon the craftsman's body parts; the width of a hand, or the length of the foot. This lead to the the cubit - the length of the forearm from the extended middle finger to the elbow. While considered a standardized measurement of approximately 500 mm, it varied by as much as ten percent between different cultures.

Accuracy or Precision
While often taken as a synonyms, accuracy and precision are different. Accuracy is an indication of how close a measure - or the average of many measurements - are to the actual value. Precision is a measure of how close the multiple measurements are to each other.

These measurement tools lacked precision. One carpenter's foot was significantly different from another's. Even a skilled craftsman would be inconsistent from one measurement to the next. As a result one of the oldest, simplest, and possibly useful computer was invented - the ruler. A ruler provided a means of precision in measuring linear distances. Rulers also resolved the problem of accuracy in measurements. As far as Five thousand years ago rulers in use were accurate to within a sixteenth of an inch.

Of course the computer that we call the ruler - even a homemade ruler - was fine for precision linear measurements. But the ancients were fascinated with angles as well. An ability to measure an angle would open up their study of the movement of the earth, the planets, and the stars. Thus the next computer was created - the protractor. The protractor made it possible to measure distances along an arc. They could then measure movements of planets and stars.

Combining the protractor with a stick and early people created the first chronograph - a sundial. The sundial is a computer that turns the position of the sun into the time of day. Computers had gone from devices that made static measurements to a device that was dynamic in that it provided updates automatically as the day progressed.

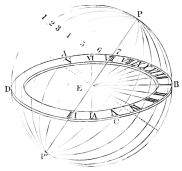


Figure 1.2: Design of a sundial using the angular position of the shadow cast by the gnoman to indicate solar time.

1.3 Chronometry

Sundials were not only time pieces. They were also calendars in that the length of the shadow formed changed throughout the year. By using a ruler and a protractor the ancients turned a clock into a means of computing the time of year and also the position of the sun and thus the angle that the earth made with the sun.

Early astronomers made use of protractors as computers that can measure the positions of the stars and the movement of the planets. And while their sextants may not have been mechanical, the data that they collected was used in what is considered to be the first mechanical computer - the Antikythera Mechanism.

1.4 Mechanical Computers

The static computers developed in antiquity were meant as measuring devices, but they were in their own way computers. They turned data in to information. But the industrial revolution brought about increasing demands for data collection and analysis.

Antikythera Mechanism

The Antikythera Mechanism is a clockwork mechanism that through the use of gears will show the position of the moon,

sun, and planets. It was discovered in 1900 in the wreck of a Roman galley that sank about 70 BCE. It is thought to have been invented by the Greek engineer Archimedes, and is now considered to be history's first mechanical computer (figure: 1.3).

Figure 1.3: The Antikythera Mechanism often credited as being history's first mechanical computer

These first computers provided their users with a means of measuring distances, locations, and time. They were also storage devices in that they were developed to model the known universe. But they did not compute - at least not in the sense of performing various mathematical calculations of which we think when we discuss computing.

Babbage's Analytical Engine

While the techniques and devices discussed are all computers, we are interested in computers that provide computational assistance; that is that can perform calculations for us. This type of computing machine is widely attributed to Charles Babbage, a Cambridge mathematician in the early nineteenth century (figure: 1.4).

Babbage, in 1812, developed the concept of what he called the *difference engine*; a mechanical machine that he hoped could be used to solve polynomial equations. He developed a working prototype of his difference engine in 1822, but dropped the project to pursue what was to become the first modern mechanical computer - his *analytical engine*.

Charles Babbage

Ada Byron King
Countess of Lovelace

Figure 1.4: Charles Babbage (1791 – 1871) and Ada Byron King - Countess of Lovelace (1815 – 1852), Charles Babbage was a Cambridge mathematician credited with designing the first mechanical computer capable of performing calculations. Ada Lovelace worked with Babbage on the analytical engine. She is often credited as being the first computer programmer

Another mathematician, Ada Lovelace worked closely with Babbage on the analytical engine. While not given the credit that Babbage received, Lovelace was instrumental in working

on the algorithmic processes for the device. As such she is often credited as being the first computer programmer.

For the next hundred years the precursor to the modern computer was mechanization. Typewriters and adding machines were developed to support the industrial revolution. Henry Ford used automation to mass produce automobiles. The mechanical devices created to support the industrial revolution are not commonly thought of as computers, but in their own way they are.

World conflict brought further advances in computing. World Wars I and II brought with them the onset of aviation mechanical devices were developed that optimized how airplanes could be used in combat.

Synchronization Gear

Early in 1914 it became clear that a pilot risked catastrophe when firing a weapon from a moving airplane. If handheld the bullets could impact the pilot's own airplane. If mounted on the airplane it had to be aimed by the pointing the aircraft into the direction of the target. When mounted directly in front of the pilot the bullets could strike the propellor.

Engineers designed a mechanical computer called a *synchronization gear* - more commonly called the *interruptor* - that could determine when he propellor was within the weapon's line of fire. At that point in the rotation it would disable the firing mechanism (figure: 1.5).

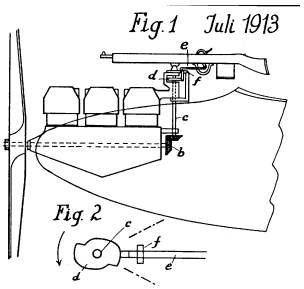


Figure 1.5: French patent drawing for a synchronization gear.

While still mechanical, the interruptor meets our definition of a computer. It collected data - the arc position of the propellor - and processed it into actionable information - a decision as to whether or not the weapon would be able to fire.

The innovation in mechanical computers brought about by the First World War pales when compared to the Second World War.

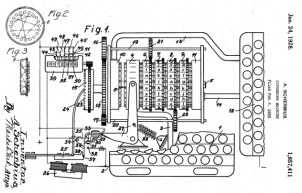


Figure 1.6: US patent drawing for the Enigma encryption device.

Enigma Machine

During the 1920s and 30s work had started on the development of advanced means for encrypting text. Generally known as *enigma*, these mechanical computing devices used a series of rotating gears and plug connectors to encrypt text. It functioned on the idea that if the number of setting permutations are large then the chance of an enemy being able to crack the cypher was very low. While the enigma machine is commonly thought of as a German device,

a United States patent was issued for an early enigma machine in 1928 (figure: 1.6).

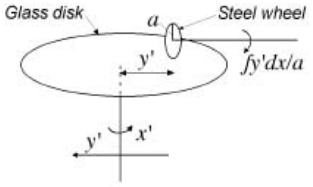


Figure 1.7: Integrator wheel for the differential analyzer.

Differential Analyzer

Mechanical computers were also developed to provide specific analyses for the war effort. In the First World War, the brand new airplane would fly at fifty knots with a maximum ceiling of several thousand feet. As the Second World War began airplanes were flying at three hundred knots and at altitudes above twenty thousand feet. A flyer could no longer simply point the plane at the target.

To resolve the issue engineers designed a mechanical computer called the *differential analyzer*. Its purpose was to solve differential equations. When first designed and implemented in the 1930*s* it was used to calculate tide tables. With the advent of the second world war it was adapted to calculate artillery trajectories.

The design of the differential analyzer was not new. It was first described in 1876. This early version was meant to perform integration, but as it proceeded it became a device to solve differential equations. It functioned by using a wheel and disk mechanism. The disk would undergo both a rotation and translation. As it rotated a wheel whose edge rested against the disk would turn. It was the rotation of this wheel that represented the output of the integrator.

The output would be a table. These tables were created by a pen drawing a curve on paper. These curves depicted the functional solution of the differential equation.

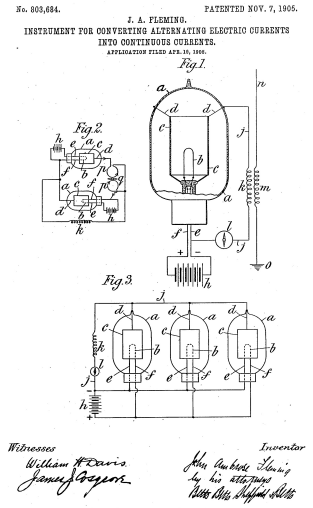


Figure 1.8: Patent application for Fleming's valve.

1.5 Electronic Computers

As the second world war came to an end, advances in electronics, beginning with Fleming's valve, the first vacuum tube, made way for the development of electronic computers. The vacuum tube provided a means of controlling the flow of current through an electronic circuit. With it, electronic devices proliferated. Sound and visual transmission through radios, telephone, and later television became common.

More directed to our interests, the vacuum tube and its advances in electronics brought with it electronic computing.

ENIAC

The first of these was the Electronic Numerical Integrator and Computer or *ENIAC*. Designed and built at the University of Pennsylvania. It was completed and began operation in December, 1945 and later dedicated on February 15, 1946. With 20,000 vacuum tubes it could perform a thousand times faster than the previous electro-mechanical computers. While the fastest computer of its time, the vacuum tubes were unreliable with several burning out every day. As such it was only functional about fifty percent of the time.

The ENIAC was developed under a war footing. Much like the differential analyzer its intended purpose was to calculate artillery firing tables for the US Army Ballistic Research Laboratory. It was used primarily for this purpose for ten year until its decommissioning in 1955.

It is often interesting to compare historical characteristics of a device with their current counterparts. The Eniac was quite imposing. It weighed about 25, 000 kilograms (27 tons) and had a footprint of 167 square meters (1800 square feet).

With regard to computing power, it could manage twenty 10 digit decimal numbers, and perform 5, 000 addition or subtraction operations per second. Much slower at multiplication, it could perform 357 full precision multiplication operations per second. It could, of course, multiply faster if less precision was required for the factors. Full precision quotients could be calculated at a rate of about 35 per second.

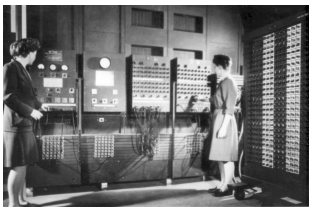


Figure 1.9: ENIAC at the University of Pennsylvania.

Compare that to a modern smartphone that weighs in at a fraction of kilogram and can compute are millions of operations per second.

Whether the computer is a two hundred year old mechanical computer or a modern electronic computer it will have a common trait. There will be the physical device - the gears or circuits - and the set of instructions - the program - that will drive the computer. The physical components are the hardware while the program is the software.

1.6 Computer Architecture

The very first mechanical computers - think back to the Antikythera mechanism - were limited by their design to

solve a single problem. Later computers - such as the Eniac and Univac - could be made to perform many different types of computation. But they were still constrained by how the operations - the programs - were entered into their computer. Those early computers, such as rulers and protractors had their operations fixed into the computer. In these cases the programming was static; once created it could not be changed. To use the rulerYou cannot change the marks on a ruler or a protractor, or add or remove planets from the Antikythera mechanism.

While these devices were programmed, the program - or the algorithm - was built into the machine. The operations were integrated into the design of the computer. The program and the hardware were one and the same.

Program
The ordered set of instructions that direct a computer on how to complete an algorithm
.

As a result, these computers did not require a means of programming them once they were built.

But some of the early mechanical computers - Babbage's analytical engine - and the electronic computers like the ENIAC and UNIVAC were *programmable* - meaning that the instructions could be changed. It could be revised, or replaced completely. This created the idea of a temporary set of specific instructions that would enable a computer to run a particular algorithm. As this was separate from the computer itself - what we now know as the hardware. Further, these instructions were more ethereal. It lacked the physical form of the computer. Since it was not hard like a physical device, it became known as *software*.

Hardware

None of us can avoid interacting with technology. We deal with a multitude of computer devices every day. On a personal level these might be phones, tablets, a laptop or desktop computer. Moving outward we use online payment systems, and ATMs. The technology that run these devices could be personal but they could be a mainframe computer that we do not even see. The common trait is this these are all examples of hardware; physical devices that run programs.

Hardware
The physical components of which a computer is made
.
"What you kick when it doesn't work.'"

But hardware is more than just the computer. It also includes all of the physical components of the computer. The processors and the peripherals such as the monitor, the keyboard, the mouse, and the physical storage memory are all hardware. We can designate these by five separate categories

Central Processing Unit	Performs the operations in the computer
Main Memory	Random access memory (RAM)
Secondary Storage Memory	Non-volatile memory
Input Devices	Devices for entering data into the computer
Output Devices	Devices for retrieving information from the computer

Table 1.1: Types of Hardware

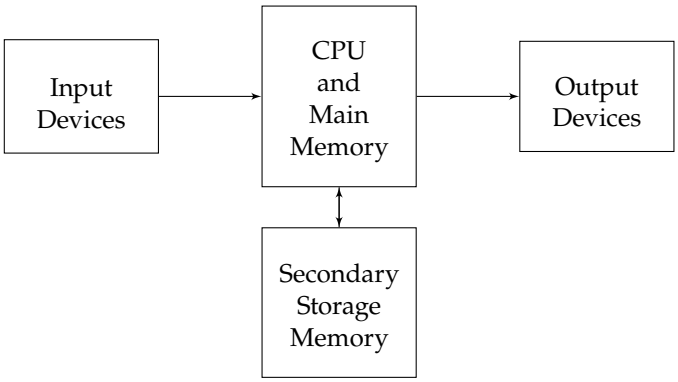


Figure 1.10: Interaction between the hardware components in a computer.

The interaction of the primary hardware components is shown in figure:1.10. The arrows indicate the direction the data travels through the hardware - *the data stream*. Data streams are both unidirectional and bidirectional. The input devices - such as a keyboard or a mouse - can only send data into the processing. Similarly the output - the monitor or a printer - can only receive data from the processor. But the secondary storage, whether it is a hard drive on the computer or cloud storage over a network are bidirectional. The data streams both into the processor from the secondary memory as well as from the processor to the secondary storage.

Software

Hardware is physical, but software is different. Hardware consists of the physical components of the computer, but software is more vacuous; it is the program. While you can read the instructions, you cannot actually hold them.

Software
The instructions that direct the operation of the hardware .

.
"What you blame when it doesn't work.'"

Software is the set of steps that we follow; what we will call an algorithm or a program. the program on a computer is the language specific set of instructions the computer will follow in completing a task. Program are commonly written in high level languages that can either be compiled to create a single

executable file to run on the computer, or can be interpreted one line at a time to complete the task.

But software can also be simple. Measuring the distance between two points involves following a set of instructions. How we do this - put the leading edge of ruler on one point and the flat edge on the other, then read the value off the rule - is the program. In this case we are the computer, our hands and eyes the input device, our brain the processor, and also the secondary memory. The process is the software.

Our goal is not to develop the hardware, but instead the software. The software can be a single line or command - or a complex set of millions of lines of instructions, but the process is the same in that it will collect data and transform it into information.

Summary

Almost anyone you ask the question "what is a computer?" will provide a similar answer; a desktop machine, a laptop, a tablet, or perhaps a phone. And while they would be correct in that they are all computers, they are not the limit of computers. Computers are simply devices with which we compute.

A computer can be a modern electronic device, but it can also be our fingers and toes. It is the ruler that we use to measure a line or the protractor to determine an angle. These devices all share a common trait. They take data and process it into information - they compute.

The computer itself can be separated into hardware and software. The hardware is the set of physical components; input devices, processors, output devices, memory. The components each perform a different task such as entering data, processing it, or delivering the information to the user - the results. To make all of the components work together requires software, a set of instructions on how to process the data.

So while computers compute, the challenge can be in how the process is done. Whether it is the task of putting a straight edge to a line or calculating complex beam loadings for a bridge, there will be a specific process we must follow. This is an algorithm. And when we implement the algorithm on a computer, it becomes a program. As we move forward we

will need to understand how to develop an algorithm and implement it as a computer program.

Self Test

1. What is computing?
2. What is the role of the electronic computer in computing?
3. What are examples of the computers used by the ancients.
4. Write about the first mechanical computer.
5. What is the Antikythera Mechanism?
6. What was Babbage's Analytical Engine?
7. Who were the first computers?
8. What is an interruptor in computer terms?
9. How did world conflict bring about advances in computing?
10. What was the use of ENIGMA machine?
11. What was the ENIAC?
12. What was the UNIVAC
13. What is hardware?
14. What are the primary components of hardware? What do each of these do?
15. What is a software?
16. Describe the difference between hardware and software. What are examples of each?

Computing and Computation 2

Computer science is not about machines, in the same way that astronomy is not about telescopes. There is an essential unity of mathematics and computer science.
Michael R. Fellows

As engineers, our interest in computer science leans toward the computer as a tool. We could argue that computer science is mislabeled. We are not interested in studying computers as much as we are in understanding how we might use the computer to model a system, or analyze a set of inputs.

For these cases, computer science would be more appropriately named *Computational Science.* We want to know how to use the computer to compute and after all, is that not what computation is all about?

So how are computing and computers different from computation? A computer is a device, usually mechanical or digital - although it does not have to be. While we commonly think of a computer as an electronic device, the computer could also be ourselves as human beings using our brains and perhaps our fingers and toes.

But computation is different from the computer. Computation is the process of making the computer, whether it is mechanical or digital, do our bidding; how we use it to perform the operations that are involved in calculating a result, or solving a problem, or modeling a system. Since our goal should also be efficiency, with computation comes the concept of complexity. Complexity relates to how long it takes to complete a task - or more specifically, how much longer it will take as the amount of data increases.

From this definition computers and computing are not synonymous - in fact they are very different. How we presently do computing can be extraordinarily complex but the idea of computing is quite simple. Computing is the process of turning data into information.

2.1 Computation

Computation is a simple concept - but far more than just adding and subtracting. It is the process of collecting data and turning it into information.In its most basic form computing or computation is synonymous with calculations such as arithmetic. While it is true that when you calculate you are processing data and are converting that data into information. But do not take the mechanics of arithmetic as absolute - computation is more than just arithmetic. It is any process that turns data into information. So while computation may involve calculations it may be something else, such as searching a database or plotting data into a graph.

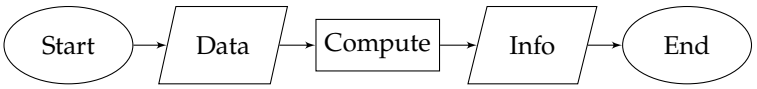


Figure 2.1: Computation is the process of collecting data and turning that data into information.

For the sake of clarity what is *data*? And what is *information*? And how are they different?

Data

Data are the basic components of computing. The data are the numbers, the values, observations and facts from which we want to gain knowledge. While we assume the data is correct it has no context. By itself, it is meaningless.

Data
Data are facts from which we want to extract knowledge or information.

As an example, data might be the number of hits, walks, and runs that a baseball scored in a game. But until it is compared to the same data of the opposing team it has little value.

Or the data is the number and weight of trucks crossing over a bridge. By itself it is important but not something we can use directly make design decisions. but if we use the data to calculate stresses or strains on the cables supporting the roadway - processing the data into information - we can decide if the cables are an appropriate size or strength.

Information

As we saw above, Information is data with purpose. That is information is also data, but it is data that can now be used for decision making.

As an example, you are given the numbers 3 and 5. This is data. The numbers are correct - as far as you know - but you cannot use them for anything because they have no context. They do not yet mean anything to you. But by the process of computing this data will become information.

Information
Information is data that has been processed, analyzed, organized, interpreted, or presented to give it purpose or put it into context. Information is intended to be used in making decisions.

Let us process this data. Five is the number of runs that the Baltimore Orioles scored in last night's baseball game while three is the number of runs that the New York Yankees scored in the same game. Further, the difference between the two values is two and the larger of the two values was assigned to the Orioles. Now the data has become information. The information is that the Orioles won the game and did it by two runs. A strained example, but it follows the process of computing - transforming data into information.

The data that was entered into our system now has meaning. Computing, or computation, turned the data into meaningful information.

And with that meaning we can use our new found information to make decisions. This is computation at its most basic.

We usually think of data and information in terms of numerical values, such as the stress on a beam or the time until an event occurs, the definitions do not require us to force numerical values onto the data. We can just as easily manipulate nominal data - that is data that is presented as descriptions or names. As an example, a set of data might include *Calculus, Statistics, Mechanics, Physics*. The respective information that we join to this data is that these are required courses for an engineering student.

So if computing is turning data into information, what are computers? Computers are the tools with which we do computing.

The early static computers - such as rulers and protractors - were used as computers to increase precision.

For example, data may be the distance between two vertical sides of a door. A ruler turns this distance into a precise measurement - say 1.2 meters. Knowledge of the distance makes it possible for us to decide if we can move equipment through that door.

Mechanical computers did something similar. The Antikythera mechanism used the data in the form of the

An argument has been made that the Antikythera mechanism was a computer that instead of converting the time into the planetary locations, it did the opposite. The user would set the dials to the current locations of the planets and then be able to read the time.

current date and time and processed it into the locations of several heavenly bodies.

Whether the computer is our fingers, a rule, a mechanical computer, or an electronic computer, there is a process that we must follow to perform computing. This process is known as an *algorithm*.

2.2 Algorithms

Every computing process requires following a set of instructions. For example our original idea of computing as basic measurement can be formalized as

```
1 Start
2 Place ruler at the leading side of the opening
3 Mark on the ruler the opposite side of the opening
4 Read width of opening off of the ruler
5 End
```

Figure 2.2: Algorithm to measure an opening

Algorithm
An algorithm is a clearly defined, finite set of instructions that when followed will perform a particular task.

The set of instructions that you follow to perform computing - or solve a problem - is an *algorithm*.

An example of an algorithm is the sequence of steps that you need to add two two-digit numbers. This is the process - or algorithm - that most children are taught early in elementary school.

```
1 Start
2 Add the two right most digits
3 If the sum is less than ten then record the sum and set carry
      to zero
4 If the sum is greater than ten then record the right most digit
       of the sum and set carry to one
5 Add carry and the two left digits
6 Concatenate this sum with the previous sum.
7 End
```

Figure 2.3: Traditional Algorithm to add two numbers

The algorithm in figure 2.3 may be the common technique but like many algorithms there are other methods as well. A second algorithm - figure 2.4 - for adding numbers was often used by bookkeepers when adding receipts.

Both techniques accomplish the same task - adding two numbers together. But they do it in distinctly different ways.

```
Start
Add the two left most digits
Multiple the sum by ten
Add the first of the right most digits to the current sum
Add the second of the right most digits to the current sum
Record the value
End
```

Figure 2.4: Bookkeepers Algorithm to add two numbers

This does not invalidate that these are both algorithms. There is no requirement that there can be only way to complete a task.

What is important is that if many different bookkeepers follow the same algorithm on the same set of data they will calculate the same sum. No matter who, or what computer, follows the algorithm if they have the same data they must get the same result.

Every algorithm follows the same criteria.

1. It shall have a single, clearly defined starting point.
2. Each step is deterministic.
3. It will reach a clearly defined end after a finite number of steps.
4. It will account for any contingencies.

Why these criteria are important can be explained with an example of ten people traveling to the same location.

1. A single, clearly defined starting point. Everyone who follows the algorithm must start at the same spot. What would happen if our ten people all had identical directions to follow, but each one started at a different location? They would start out going the same direction and turning at the same times, but they would be at different locations when they do each task. The result is that they would finish at ten different places.
2. Steps are deterministic. What happens at each step cannot be left to chance. If different people follow the algorithm they must each have the same result. In our traveling example, each person who arrives at an intersection must make the same turn. If some randomly decide to turn left, while others turn right or continue straight, the again would finish at different locations.

3. Ends after a finite number of steps. The algorithm cannot be open ended. If the traveling directions had a set of four right turns it is possible that our travelers would go in circles forever - an infinite loop.
4. Accounts for any contingencies. What if during the trip there is construction and the road is blocked? The algorithm must be able to address necessary alternatives. This issue of contingencies brings up the possibility that for a particular set of data there is no solution. In that case the algorithm should allow for an end. If our travelers are flying, and the airport is socked in with fog, then their flight is not leaving. They have a choice to make - find alternative transportation or just go home. If they go home, then the algorithm ends despite not providing the desired result. In this case the algorithm has two ends - but still only one start.

An algorithm is a precise set of rules that describe the specific steps that will solve a problem. If you implement the algorithm correctly you are guaranteed to find the solution to a problem.

Program
A program is an ordered set of instructions that implement an algorithm.

So how do we implement the algorithm? It is generic. It is not designed to be implemented by a specific person or on a specific computer. To do this we need to convert the algorithm to a form that can be understood by the device. We need to write a program. Thus, to implement the algorithm we run a *program*.

In this case the program is a *computer program*; a program that is designed to be run specifically on a computer. But programming is a general term that involves any type problem solving. In fact, programs are the general name for any ordered set of instructions that need to be followed. This could be a computer program, or - from the field of optimization - what is known as mathematical programming, linear programming, or dynamic programming. Or, quite simply, a program is the list of names that present the order of performance of children at a piano recital.

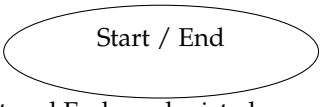
Semantically we often interchange the terms algorithm and program, But there is a slight difference. While the algorithm is a process that performs computing, the program is the specific, ordered set of instructions that when implemented will solve the problem.

The algorithm lays out the steps involved in the process. While we could just write out each step - this is often known as *pseudo code*, a common technique is to create a schematic of the program - a *flow chart*.

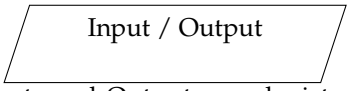
2.3 Flow Charts

A flow chart is a schematic of an algorithm. It shows each step involved in the process with connections indicating the next action once a step is complete.

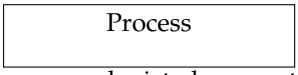
Flow charts are commonly written with a set specific geometric shapes for each the type of the action. Ellipses are used to indicate the Start and End; trapezoids for inputs and outputs, diamonds for decisions, and rectangles for executable - or action - steps.



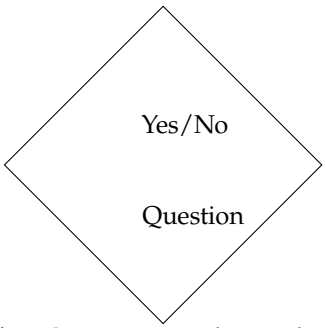
Start and End are depicted as an ellipse



Inputs and Outputs are depicted as a trapezoid



Processes are depicted as a rectangle



Yes/No Questions are depicted as a diamond

Figure 2.5: Flow Chart Symbols

While there are many other symbols that used in flow charts, these four are the primary ones and the only that we will use.

In a flow chart each node connects to one other node. These connections are unidirectional and one to one - that is they may only connect one node with one other node. You cannot create a one to many node connection. Doing so would violate the rules of algorithms in that each action must lead to the same next action each time.

There is one action that at first appears to violate this rule - the decision diamond. When we implement the decision

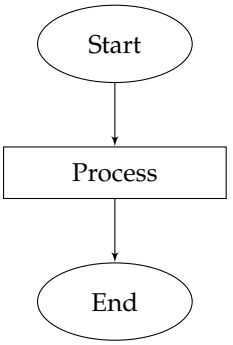
there will be exactly two paths; one for true or yes, and a second for false or no. This is still deterministic since there is only one path if the answer is *yes* and it is always the same path. The same holds for when the answer is *no*.

With one exception any descriptions are written within their shape. The exception is the connection labels for the decision diamond. Convention is that a small *true* or *false* be written next to the path. These labels are not actions so they are not placed inside of a shape.

There are three different structures that we will be using in flow charts and in our programs; the *Sequential Structure*, the *Selection Structure*, and the *Repetition Structure*.

Sequential Structure

The most simple algorithm to be depicted in a flow chart is a sequential structure. A sequential structure is when each action in the algorithm follows directly from one to the next.



"Begin at the beginning, the King said very gravely, and go till you come to the end: then stop."
Lewis Carol in Alice in Wonderland
Figure 2.6: Flow Chart for Sequential Structure

In a sequential algorithm the order of the actions are important. If you change the order then the results you would expect the results to be different.

Sequential Structure
A sequential structure is a programming structure in which each step follows a linear path from start to end.

In its most simple form, figure 2.6, the sequence is *Start - Process - Stop*. This is linear - each step follows the same one that immediately precedes it. If you run the program multiple times it will follow the same steps in the same order every time.

We have already seen a flow chart that implements a sequential algorithm. Figure 2.1 depicting the process of computing - turning data into information - is a sequential structure.

Another example is the bookkeeper's algorithm (figure 2.4). The algorithm can be written as a flow chart using a sequential structure (figure 2.7).

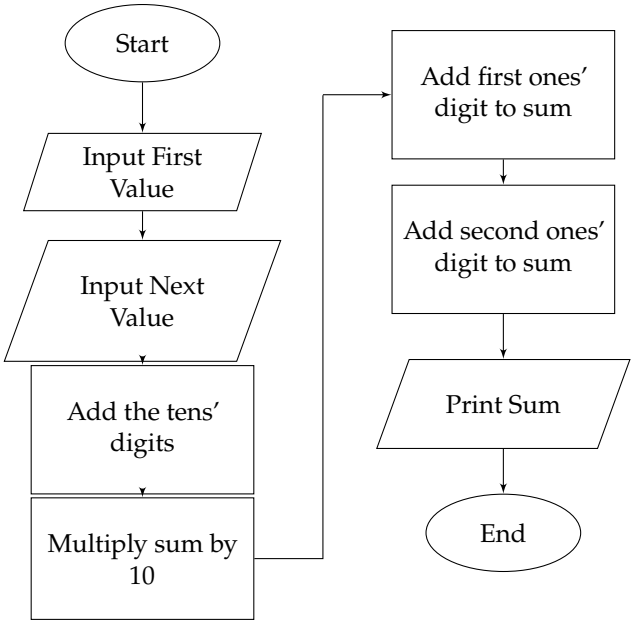


Figure 2.7: Flow Chart for the Bookkeeper's Algorithm

Recall that the sequential structure is linear - it never leaves the same path. But what if there are exceptions? Perhaps a number is negative in which case you should subtract the digits. This requires a second structure - a *selection structure*.

Selection Structure

The sequential structure provided a means for following a specific set of steps - one after the other. Every time that the algorithm is run the results are the same. But it often occurs that a decision must be made. Perhaps a low temperature requires a furnace to be turned on. Or a pathway is blocked and an alternative will have to be followed. Or in the case of the Scarecrow from the Wizard of Oz, you simply have a preference of one over the other.

Selection Structure
A selection structure is a programming structure in which a true - false (yes or no) question is asked. The answer determines which of two paths the algorithm will take.

To incorporate this decision making ability to an algorithm requires a second structure - the *selection structure* or what is commonly called the *branch*.

An example of the use of the selection structure in the Bookkeeper's Algorithm is dealing with negative values. The sequential structure is fine when each value is positive - you

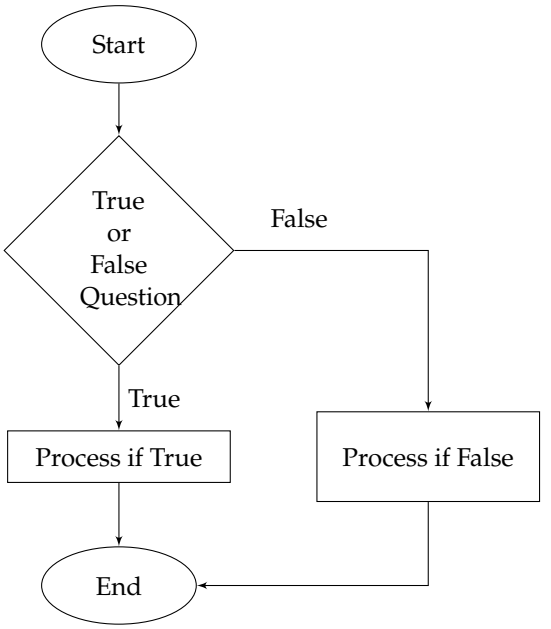


Figure 2.8: Flow Chart for Selection Structure

just add to the running total. But if it is a debit instead of receipt you would subtract it from the total. Since you would be subtracting twice - first for the tens value and then for the ones value - the algorithm needs to know whether to add or subtract. We can do this by adding in a selection.

There is an argument that it is this decision making process in the selection structure that makes a computer *reason*. While it is true that this is the foundation of artificial intelligence it is far from true reasoning. The reason for this is in the ability of the selection structure.

Warning The decision controlling the branch of a selection structure is binary. It can only take on one of two possibilities. In this any question must be answerable as *Yes* or *No*, *True* or *False*, or *1* or *0*.

The decision component in the selection structure asks a type of question; *Is the number negative?*, or *Is the fruit an apple?*. This is similar the game of twenty questions. Each question must be answered as *Yes* or *No*, or *True* or *False*, or using integers 1 or 0. That is it. They questions are not open or free response. This means that you cannot ask questions such as *What would you like for lunch?*, but you could ask *Would you like a peanut butter and jelly sandwich?*

What if we have three numbers to add, or four, or a hundred? The algorithms that we have been demonstrating sequence and branches, but what if we need to repeat? This requires the third structure - the *repetition structure*.

Repetition Structure

There is nothing that a computer program does that a person could not do themselves given enough time. The value of computing is not necessarily that the computer can calculate very quickly but that it can do the same calculation hundreds or millions of times. A person would never do that. The application of performing the same operation multiple times employs a *repetition structure*.

A computer does not do anything any of us could not do. It just does it over and over again without complaining

David Barron

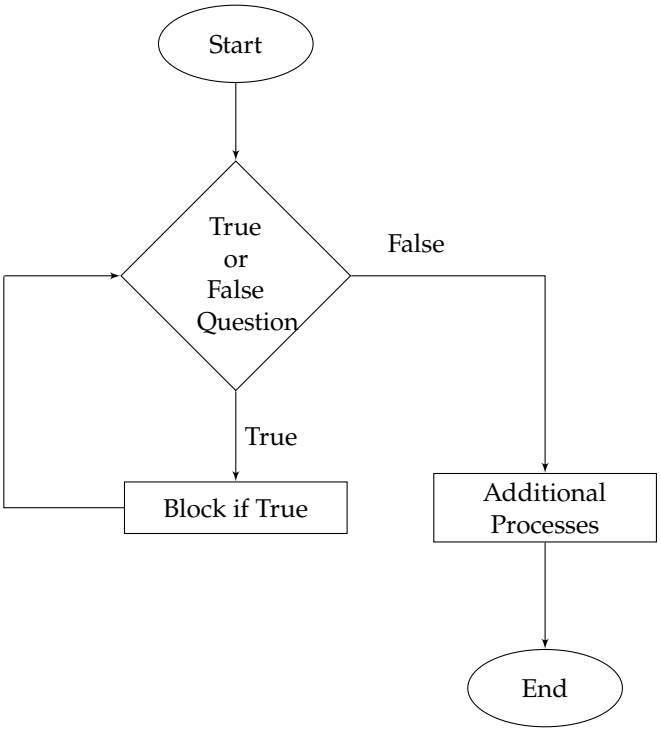


Figure 2.9: Flow Chart for Repetition Structure

Extending the addition of a sum of numbers algorithm, a repetition structure could be employed if instead of two numbers we are adding three, or four, or one hundred. We could assume that we do not know have prior knowledge of how many numbers are to be added. The repetition structure would then run as many times as we have numbers.

Repetition Structure
A repetition structure is a programming structure in which a set, or *block* of instructions are repeated multiple times.

The flow chart in figure 2.10 is shorter than th sequential approach. But more importantly it is open-ended. We could have extended the sequential structure for adding numbers to any amount, but whatever that number of values is we would be stuck with it. If we wanted to add different number of values we would have to change our algorithm.

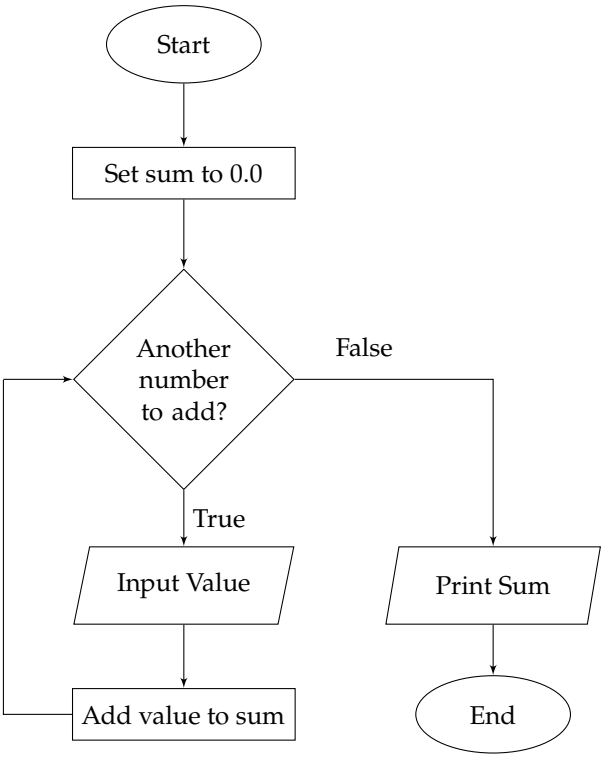


Figure 2.10: Flow Chart for Using Repetition to Add Several Numbers

For the sake of simplicity, the flow chart using repetition is for adding a set of numbers. We could change this algorithm to the bookkeeper's algorithm by adding two repetition structures. The first would be for adding the tens digits then a second for adding the ones.

We have seen three different structures for depicting an algorithm. Further we took note that if we replaced the sequential structure with a repetition we could make the addition algorithm more versatile. But we should also be interested in the efficiency of the algorithm. Is there one structure that is more efficient than another?

2.4 Complexity

An algorithm identifies each action to be taken, but as was shown there can be more than one algorithm for a particular task. So which to use? The choice of algorithm may come down to several items. Which algorithm is the easiest to

understand? Which is easiest to implement? Which is the most efficient in terms of the number of computational steps involved? Which algorithm is the most efficient in terms of the amount of space - or memory - required to implement it?

Computational Complexity
Computational complexity is a measure of the rate of increase in the additional steps as the amount of data increases.

In analyzing an algorithm we must take into account two items; the amount of memory that will be used, and the number of steps that it will take to complete the algorithm. The memory is known as *spatial complexity* while the number of steps is the *algorithmic complexity* or *computational complexity*.

The number of steps is not the same as the amount of time. The steps in an algorithm is independent of the platform on which the program is being run, while the time is dependent upon the processor speed. A program on a slow computer or fast computer will still have the same complexity.

Comparing the complexity of different algorithms is done by creating a function of the number of steps with respect to the size of the input. If an algorithm has n data inputs then the complexity would be $f(n)$. As an example, two algorithms may have two complexity functions; $f_1(n) = a_1 n + b_1$ and $f_2(n) = a_2 n + b_2$. Depending upon the values of a_1, a_2, b_1, and b_2 one of these algorithms may use fewer steps or more steps than the other.

While it is tempting to calculate the actual number of steps, or the amount of memory, the actual number of steps is not what is important - the rate of change in the number of steps is. Being the rate of change, complexity is the first derivative of the number of steps. Further, the actual values are not of interest. Complexity is a relative measure.

Table 2.1: Ordered list of algorithmic complexity

Complexity	Running Time	Example
Constant Time	$O(1)$	Calculating the median of a set of values
Logarithmic Time	$O(\log(n))$	Searching an ordered set of values
Linear Time	$O(n)$	Calculating the mean of a set of values
Log - Linear Time	$O(n \log(n))$	Comparison sort algorithm
Quadratic Time	$O(n^2)$	Bubble sort algorithm
Polynomial Time	$O(n^p)$	Set of nested repetition structures
Exponential Time	$O(b^n)$	Brute force password attack
Factorial Time	$O(n!)$	Traveling salesman problem

The rate of change is described by the term with the largest first derivative. For example, if the number of steps is $f_1(n) = an + b$ then the magnitude of the derivative is controlled by n, the linear term. Because of this the number of steps - or complexity - will increase linearly as the amount

of data increases. Another way of looking at this is to ask "How much longer will the algorithm take if the amount of data doubles? or increases by a factor of 10? For the linear algorithm if the amount of data doubles then

$$\begin{aligned} s_2 &= f(2n) \\ &= 2an + b \\ &\approx 2f(n) = 2s_1 \end{aligned} \tag{2.1}$$

which is approximately double the number of steps for $f(n)$. If the amount of data increases by a factor of ten then the number of steps is about ten times the number of steps for s_1. The complexity for this algorithm is described as $O(n)$ - read as "Big O of n". $O(n)$ is also known as linear time. This means that the number of steps is a linear function of the amount of data.

Constant Time

The fastest algorithms operate in $O(1)$ time; known as constant time algorithms. While it is often erroneously thought that the algorithm takes a single step, it in fact takes a constant number of steps. This constant could be any value. As a function, if there are n items of data the algorithm takes A steps. In this case increasing the amount of data - have the amount of data grow from n to $n + 1$ - the number of steps does not change. It is still A.

Median
The median is the value that divides a sorted set of data into two equal sized halves. If the number of observations is even then fifty percent of the observations are below this value and fifty percent are above. If it is odd then the two halves consist of fifty percent of the observations not including the median value itself.

An example of a constant time algorithm is determining the median of a sorted set of values. If you have a list of 5 values and they are sorted from the smallest to the largest then the median value - the value at which half of the data is below and half is above - is $\hat{x}$ where

$$\hat{x} = \begin{cases} \frac{1}{2}\left(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}\right) & n \text{ is even} \\ x_{\frac{n+1}{2}} & n \text{ is odd} \end{cases}$$

The formula for the median shows that the number of observations - the data - is not a factor in the algorithm; only knowledge of the value for the number of observations is. If $n = 5$, or 10, or 5000 the number of steps remains the same.

Example:
A class of forty one students are sorted by their height. Each one is sitting in a numbered seat - say from 1 to 41. The professor wants to find the the student with the median height.

The median will be the student in seat $\frac{41+1}{2} = 21$. The professor asks student twenty one to stand.

Now there are 287 students again sorted and in numbered seats. The median height student is the one in seat $\frac{287+1}{2} = 144$. It took the same number of steps despite there being seven times the number of students in the data set.

Logarithmic Time

Between constant and linear time is a complexity known as logarithmic time, $O(\log_2(n))$. An example of a logarithmic time algorithm is a binary search. A binary search is done on an ordered set of data. To find the location of a particular key the algorithm first checks the middle most value. If the key is smaller than the middle value then all of the upper half is ignored and the binary search is repeated on only the bottom half. This is repeated until the value is found or there are no more elements in the set.

Binary Search
A binary search algorithm is one in which a sorted set or list or array of data is searched by checking the middle most observation. If the comparison does not match this value then either all data below or above is excluded from the search and only the remaining half is searched. The binary search has a complexity of $O(\log_2(n))$.

The speed of the the logarithmic time algorithm is easily calculated. Since the binary search eliminates half of the remaining data each step, a data set with $n = 32$ will have $n_{mboxremaining} = \{32, 16, 8, 4, 2, 1\}$ with each pass. This means a worst case search time of $\log_2(32) = 5$ passes. A recursive form of the binary search algorithm will be presented in chapter 8.

Linear Time

As mentioned before, an algorithm that is slower than logarithmic time is linear time, $O(n)$. This is a common complexity for a single repetition structure or loop.

Mean
The arithmetic mean of a set of data is the sum of the values of each observation divided by the number of obervations.

An example of a common linear time algorithm is the calculation of the arithmetic mean of a set of data.

$$\bar{x} = \frac{\sum_{k=1}^{n} x_k}{n}$$

This calculation can be done using a single repetition structure. For each additional observation the algorithm will require an additional step, thus $O(n)$

Log-Linear Time

There are many algorithms for sorting data. While it is possible in a specific case to sort a set of data in linear time, sorting algorithms are commonly slower than $O(n)$. The fastest of these is the comparison sort.

A *comparison sort* functions in $O(n \log(n))$ or what is known as *log-linear time* or *sub-linear time* because while slower than a linear algorithm it is still faster than $O(n^2)$.

Quadratic Time

A second sorting algorithm - and one that is commonly taught because of its ease of programing - is the *bubble sort*. It runs in $O(n^2)$ or *quadratic time*.

The bubble sort is an example of an algorithm that is implemented using two nested - one inside of the other - repetition structures. Two nested loops, running in $O(n^2)$ time, is an example of a *quadratic time* algorithm.

Polynomial Time
An algorithm is polynomial time if the number of steps to run the algorithm is $O(n^p)$.

Polynomial Time

The quadratic time algorithms are members of a set of algorithms know as *polynomial time* algorithms or $O(n^p)$ where $p > 0$ is a constant.

Any algorithm that is polynomial time or faster is considered operational. That is it can be programmed and for increasingly large amounts of data can still be run to completion. This may appear paradoxical since there is not upper bound on p. Clearly there is a difference between a $O(n^3)$ algorithm and one that is $O(n^{100})$. And there is, but they both will run to completion.

The issue - and a reason for our interest in algorithmic complexity - is that of algorithms that run in *non-polynomial* time.

Exponential Time

A common issue confronting all of us is security. Let us start with a simple example. A combination bicycle lock takes four digits. Since there are ten possible digits for each there are $10^4 = 10,000$ different combinations. A computer algorithm to crack this would only need to try $10,000$ different combinations.

But what if instead of four digits we require eight? Then the growth in the number of steps to crack the lock just increased to $10^8 = 100,000,000$ different combinations.

The growth in the number of combinations that the computer would need is an example of exponential complexity. In this case it is $O(10^n)$. The general form for exponential complexity is $O(b^n)$ where b is an unspecified constant.

It is the challenge of exponential complexity that can make computer passwords more secure. While many companies require passwords to be a minimum of eight characters with mixes of upper and lower case, numerical digits, and characters, these passwords still come under the cutoff for a fast computer to crack - and are of course very difficult to remember.

The alternative may not be to make the password more complicated by mixing letters and characters, but simply to make it longer. If the only requirement is that the password be at least twenty characters long, then there are $26^{20} = 2x10^{28}$ different passwords. And while twenty is not actually all that long the exponential complexity of a brute force attack makes it difficult to break.

Another example of an exponential time algorithm is the recursive Fibonacci function. This algorithm is $O(2^n)$. We will look at this in chapter 8.

If n is small, a $O(b^n)$ algorithm can be completed. But recognize that for each addition data point the number of steps increases by a multiple of b.. This rapidly increase in the time makes any algorithm with a much more data completely unusable.

But while an exponential time algorithm is unusable for all but the smallest data sets, there is a common time complexity that it even worse.

Factorial Time

There is a famous problem in which a traveling salesman must visit n cities. The goal is to find the path that connects all of them that has the minimum distance. The brute force approach is to start a city then go to every city, recording the distance. You then move to a new starting city and repeat this, continuing on until you have travelled from every city in the network to every other city. If there are n cities then this

Traveling Salesman Problem Given that there are n cities with known distances between each city, what is the shortest possible route that a person can visit each city and return their starting point? The Traveling Salesman Problem - TSP - is a common example of a factorial time algorithm.

algorithm will take $n \cdot n - 1 \cdots 2 \cdot 1 = n!$ steps. As such the algorithm is $O(n!)$ known as factorial time.

The complexity - or running time - of an algorithm is important not for what it tells us about the actual number of steps, but for when it tells us that an algorithm is unusable. While polynomial time algorithms may appear to be inefficient - $O(n^50)$ is a polynomial time algorithm - they can still be coded and run to completion given a reasonable amount of time. So even as the amount of data increases the number of steps also increase but not so fast that the time grows out of control.

In reality, an algorithm that is $O(n^50)$ is an outrageous number of steps but it is still polynomial time. The reality of programming is that in engineering it is unlikely that we would ever have a polynomial algorithm that is slower than $O(3^n)$. This would be the complexity if you have three loops nested one in inside of the other. You might have this if you were programming a finite element analysis in three space.

While polynomial time algorithms tend towards $O(n^2)$ and possibly $O(n^3)$, there are common models that go beyond the polynomial time into exponential and factorial time algorithms. Unless the size of the data set is extremely small the amount of time that these algorithms would take to run to completion would often be beyond our abilities - for even the fasted computers. As a result there are many non-polynomial algorithms in which active research is to develop an alternative that will run in polynomial time. Until that time, if you know the algorithm is exponential or factorial it is probably best to step back and look for an alternative model.

2.5 Engineering Design Process

Creating a flow chart is a step in the process of creating an algorithm. But most of the time our goal is a bit more. We want to create a computer program to assist us in an engineering analysis, or in modeling a system, or as part of a design process.

Too often when writing a computer program we take an approach similar to one (and just as wrong) as we do when writing a composition for an English class. The professor tells us all of the steps involved in creating writing such as brain

storming, outlining, writing rough drafts, and so on, but in the end we just start writing the composition.

Programming is similar in that when we have to write a program we just start writing it - without much thought to how or why. Instead we can follow the standard approach of an engineering design.

There is a process that we, as engineers, use as part of the design process. We

- Define the problem
- Specify the requirements
- Build a prototype
- Build the product
- Test the solution

The engineering design process is directly adaptable to writing computer programs. It does not matter if we are writing a short code or building a large software project, the process is similar - just a matter of scale. Stating the problem is exactly the same. The specifications and requirements for a design project become understanding the inputs and outputs of the program. Building a prototype becomes working out a solution by hand. The final product is actually the most similar - building the product and writing the program are the same idea. A flow chart of the engineering programming design process is shown in figure 2.11.

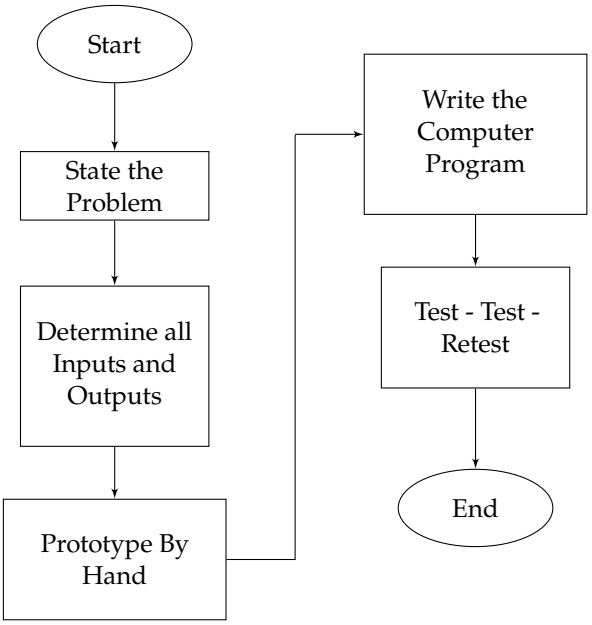


Figure 2.11: Engineering Design Process

Whether we are building a bridge or designing a manufacturing line the five steps are self explanatory. But what do they mean with respect to writing software?

State the Problem

The first step is also the one most likely to be overlooked. We are already convinced that we know what the program must do, but do we? A statement of the problem to be solved can often provide a path that will need to be followed, including the many steps that might be overlooked if you had just started to write the program. The simple step of stating - and perhaps outlining - the problem can solve hours, days, or weeks of adding new code or rewriting the code after the fact.

Determine the Inputs and Outputs

Much like the previous step of stating the problem, we often jump into coding without a clear idea of what data will be needed to run the program and what information will be created when it is done. We will as we develop programming techniques that a common style is is to group all of the inputs and later the outputs together. If we are clear at the start what inputs and outputs we will need we can start our program by creating the necessary code to enter the data and return - commonly print - the information. With this done we will know what we call all of the inputs and outputs, and as important we do not have to add input or output code as we developed the program.

Write the Program

It took three steps but you are finally at the point where you would actually write the computer program. If you have followed the process this step should be direct - you have all the details that you need for your code.

There is also are also possible sequences that you can follow in writing your program. By having the plans in place - perhaps creating a flow chart you could save hours or days on coding.If you are working alone then you will be responsible for the entire program. In this case there is an approach to coding that can be beneficial.

There is a paradox in computer programming. It is *where do we start?* The conventional wisdom would be to start at the beginning just like we did when designing an algorithm and creating a flow chart. But this is actually backwards. Much

Teaching a dog to swim If you want to teach a dog to swim you need to address the question *what do I teach first?* The common responses are to teach them to get into the water, or to dog paddle, or just to float. In fact the first lesson should be on how to get out of the water. If the dog cannot get out then all of the lessons are for naught.

like teaching a dog to swim we need the end results first. Thus we do not start with creating inputs or even calculations, but instead we start by creating the output.

The first step in writing a program is to return all of the outputs. It does not matter that you do not have any input data or calculations or the information that you will be returning or printing. Make up output information that will act as a temporary placeholders until the real information is available. You can even go so far as to work in all of the formatting so that the output - when it is using real information - looks just the way you want it.

The next step is actually the middle step. Now create fake input data and work on the computations or analysis.This way you have consistent inputs and do not have to repeatedly enter them into the program. If they are hardcoded they will not change and you can save hours entering and reentering the same data.

The final step in writing the program is to create the start - the data input. Since the rest of the program is done all that is required here to create the interfaces for data input. This could be by having a user enter the data, or by having the program read or access the data from some other source. What is important in this step is data integrity. Is the data that is being used correct? This is where all of the initial data checking and verification will take place

Once the data input is complete, is the program done? Sadly, no. Now comes what may be the most challenging step of all - Quality Control. You need to start testing.

Test - Test - Retest

If the program has been written, why is there an additional step? The reality of any computer program is that if there is a problem with it, the programmer will be blamed. And the chances are always good that there is a problem. Thus the quality control step for even the most simple program.

When you test you need to perform multiple tests. This usually involves changing inputs to the model. You need to test the common - the inputs to the program would be expected. For example, if the model expects temperatures to be between twenty and one hundred degree, then run the program with many values between twenty and one hundred

degrees. The goal in this case is to determine if the program works as expected when it is used as expected.

But you should also test it against the extremes. While twenty and one hundred might not be expected, they are still possible - albeit rare - and should be tested as well.

But the third case is that of nonsense. If the input can never be below twenty, then test values of zero, or negative values. The same applies to the other other extreme. If the maximum possible value is one hundred then it is important to also check one thousand, or one million. In these cases the program should probably catch the input error to ensure that the input could not happen. While these values should never be entered into the program under normal circumstances, a user could enter them by mistake. Since the user did not enter these values on purpose, the chances are great that they would not know that they had made the error. Thus it is up to the program deal with this type of error.

2.6 Programming Errors

Testing is meant to identify errors in the program so correcting the errors would be the natural progression from testing. But correcting an error requires knowledge of the type of error.

Programming errors can be classified as three types; *syntax errors*, *run-time errors*, and *logic errors*.

Syntax Error

The first type of possible error is the *syntax error*. This is commonly thought of as a typo, or a misspelling. It occurs because the program statement has been written incorrectly. As such the computer cannot process and execute the statement. Since the individual statement cannot execute, a program with a syntax error will not run to completion.

Syntax Error A *syntax error* is an error in the way a programming statement is written.

Syntax errors do not require much more testing than trying to run the code. If the program does not run then there is a good chance that there is a syntax error. Most compilers and interpreters will indicate line where the syntax error is. Common syntax errors occur when a variable name or a command is misspelled. Another might be missing a close such as a } or the word **end** on a block of code.

Runtime Error

Runtime Error A *runtime error* is an intermittent error that might result in the program ending prematurely with some runs but not with others.

Syntax errors are identified when the user attempts to run the program. But the run-time errors are more troublesome. While they still result in the program not running to completions, they are intermittent - sometimes the program runs, but other times it does not. There are many possible reasons for the runtime error. Perhaps there is a line the program in which the square root of a value is calculated. Most times the input to the square root is non-negative and the program calculates the square root and continues on to the end of the program. But there are some inputs that will result in a negative value being sent to the square root function. If the program does not address the imaginary value from the square root, it will crash the program.

To identify and correct the possible run-time errors you need check your code against many different inputs. Remember that the goal is to make the program fail. As such try inputs that might result in a divide by zero, or a negative square root, or create an infinite loop. These inputs may not exist in reality but as long as the program allows them there is possibility of run-times errors.

Logic Error

Logic Error A *runtime error* is an error in the way a programming statement is written.

The main reason for testing is the logic error. Recall that a logic error is when everything runs but the results are just plain wrong. In this case there is no indication of a problem so the user will just take the information and run with it.

Black Swan Effect
The black swan effect is described as some event that is so rare - or even impossible - so little attention is paid to it. Then when it happens it has a major impact on the system. It was described by economist Nassim Nicholas Taleb

The Black Swan Effect describes an event that is unexpected but then happens - often with severe consequences. It comes from the idea that since no one had ever seen a black swan, they therefore do not exist - but then suddenly one appears. It is a means of explaining the importance not ignoring events that have a low probability. It is a commonly discussed in finance and investing but can be just as important in engineering.

The approach to eliminating logic errors is to try many expected inputs. If the range of input values are fifty to one hundred try runs with input running the full range - all with the goal of trying to make your program crash. And remember the limits. If the normal range is fifty to one hundred but can be as low as zero or as high one thousand, check fifty and check one hundred. But also run your program with zero and one thousand. If the extreme inputs

can happen they eventually will happen. Remember the *Black Swan Effect*.

But we also get into the nonsense. There is a saying in programming that *you need to make your program idiot-proof. But remember that once you do, someone will invent a smarter idiot.* This means that if you have every reasonable input covered there is still the chance that someone will enter unreasonable data. You need to test these values as well. If your program is well written then error checking the inputs will catch even the most ridiculous inputs.

Chapter Summary

Computing is the process of converting data in to information. Not a complex idea but one that defines a clear goal. Data are facts. But these facts are the basis behind making decisions. Decision making requires information. To transform data in to information we compute.

This process commonly follows a specific steps that must be done in a predetermined order - an algorithm. An algorithm is the process. With a single fixed starting point and a set of pre-determined steps it will perform this transformation. The magnitude of the number of steps that the algorithm needs is measured by its complexity. For a person the complexity must be small, constant or linear for most data. But computers can automate the process and enable us to perform millions of calculations a second. This opens up the computation to algorithms that operate in quadratic or more generally, polynomial time. But the growth rate of a non-polynomial time algorithm, such as those that run in exponential or factorial time, will be unusable on even the fastest computers for all but the smallest data sets.

The algorithm is more commonly known as a program regardless of whether or not it is run on a computer. But our goal is to automate the process and have a computer do the processing for us. This requires us to program the computer to perform the algorithm.

The process of writing the program can be approached in a manner similar to any engineering design process. We state the problem. We determine the requirements - in this case the inputs and the outputs. We prototype the solution, or in programming we work out a simple solution by hand. We

build the product or system - we write the program. And finally we test, and test, and retest.

Testing a program is intended to identify two of the three types of errors. Errors can occur as syntax errors, runtime errors, and logic errors. Syntax errors, such as misspellings, or incorrect statements, will result in the program stopping at that point. It will never continue on past that. But runtime errors and logic errors are different and the reason it is so important to thoroughly test a program.

A runtime error will sometimes run but it may also crash at a point. Because it may be intermittent it is important to test a wide range of inputs. The more varied the inputs the more likely the runtime error will occur.

A logic error occurs when the program runs but returns incorrect results. The hand solution is always a good starting point in identifying logic errors, but it is just as important to test impossible inputs to confirm that impossible inputs do not result in what appears to be possible results.

Programming as a method of problem solving or decision making does not require a computer, using a computer to solve the problem is faster, can be used on models too large to be done by hand, can be repeatable without careless occurring, and is simply more fun.

But to program the computer will require a method for entering the program into the computer. For this we will need a programming language.

Self Test

1. What is a computer? How is it different from computing?
2. What is data? What is information? Give differences between them.
3. What is an algorithm?
4. What is a basic schema of an algorithm?
5. What is an algorithm?
6. What is the difference between an algorithm and a program.
7. What is flow chart and why are they used? What are all the shapes used in a flowchart?
8. What are the different flowchart structures?
9. Provide an example for sequential structures?

10. Write a flowchart for multiplication of two numbers using repetition structure.
11. Why is complexity of an algorithm important?
12. Explain the different time complexities. What is the best and worst complexity for an algorithm. Illustrate a case where polynomial complexity can be considered worse than an exponential.
13. Explain how a binary search has logarithmic time complexity.
14. How are exponential algorithm useful in secure password mechanisms?
15. What are steps involved in engineering design process
16. Explain the ideal process of writing a program
17. What are the different programming errors? Explain each of them. How are they the same? How are they different?

Programming Languages 3

An algorithm is a set of steps - instructions - for a program to follow. What runs the program is not really an issue. The algorithm could be implemented by a person - something that we do every time that we provide directions. Or it could be a machine, in which case the machine is now a computer. In either case when we speak of the computer we are speaking of *hardware* - the physical components of the computer.

But we need to direct the hardware on how to implement our algorithm. We do this by providing a *program*. The program is what we commonly think of as *software*; the instructions that make the hardware perform for us.

For a person we just tell them the steps that we would like them to follow. By doing this we have in effect programmed them. But what about a machine? How do we program a machine?

3.1 Low Level or High Level Languages

The process of programming the early mechanical computers required adjusting gears or levers. The electronic computers replaced the gears with wires or switches. In either case it was a physically intensive process. It was also a time consuming process in that it could take days or weeks to program the computer to implement an algorithm.

Software
The instruction set that controls the computer's hardware
.
"What you blame when it doesn't work.'"

These programs were also temporary. Once the programming was complete the algorithm could be run. When it was complete the programmers would reset the computer and start the process of programming the computer for the next algorithm needed.

This type of programming was often done using wires and switches. It could take weeks to program the computer for a single algorithm. Once programmed It might return a result in minutes, but then the programmers would start the process of rewiring the computer for the next program. If an engineer wanted to run another case or adjust a model they might have to wait weeks until once again it was their turn on the computer.

Portability
The ability for a program to be run on multiple computers without changes
.

This was a result of the computer's design - it was all hardware. All of the tubes, wires, transistors, and circuits were a single unit. To program the computer the programming would determine how the circuitry would interact and then ent enter the program using switches and cables. The commands were written directly in what would be considered *machine code*.

The speed and complexity of programming the computer was a major problem. But it was not the only one. The software lacked portability.

When a programmer created the algorithm for their analysis it was written for the single computer upon which it would be run. They could not write it once and then run the program on different computers. It lacked portability. The program was, in effect, a *one-trick pony*.

One-Trick Pony
Someone or something that is only good for one particular purpose, or at doing one particular thing
.

To speed the process of programming, and at the same time make the program portable, the software needed to be separated from the hardware. This would require a separate means of writing the program, loading it into the computer, and then having the computer run it. This was accomplished by the creation of a standardized form for writing the instructions. This list of commands could then be used to program the computer quickly - at least much more quickly than what was currently being done. Along with this came repeatability and portability. By using this formalized set of instructions a programmer could enter their algorithm into the computer - or any computer that would accept the same commands - and run the program. And if necessary, it could run it over, and over, and over again.

This was the first use of a programming language.

Low-Level Languages

The first programs were implemented in a *low-level language*. The most common low-level programs are written directly in machine code or a series of binary, 0s and 1s, instructions.

Low-Level Languages
A programming language that contain instructions that are directly readable by the computer

A low-level language is directly readable by the computer which makes it difficult for a programmer to understand. The benefit is that the programs often run very efficiently; both fast and using much less memory.

While efficient, low-level languages are difficult in which to write programs. One type of low-level language, machine code, requires the programmer to provide the instructions to

directly control the computer's Central Processing Unit (the *CPU*.) This might involve the direct allocations controlling individual memory addresses or performing specific tasks such as loading a value or performing an arithmetic operation. Machine code might be written in the binary code that will be executed by the CPU.

An example of the machine code that would print *Hello, world* is shown in figure 3.1. While this example is not binary, it instead operates by allocating, assigning, and accessing individual memory locations by their address.

```
1  b8  21 0a 00 00     #moving !\n into eax
2  a3  0c 10 00 06     #moving eax into first memory
       location
3  b8  6f 72 6c 64     #moving orld into eax
4  a3  08 10 00 06     #moving eax into next memory
       location
5  b8  6f 2c 20 57     #moving o, W into eax
6  a3  04 10 00 06     #moving eax into next memory
       location
7  b8  48 65 6c 6c     #moving Hell into eax
8  a3  00 10 00 06     #moving eax into next memory
       location
9  b9  00 10 00 06   #moving pointer to start of memory
       location into ecx
10 ba  10 00 00 00     #moving string size into edx
11 bb  01 00 00 00     #moving stdout number to ebx
12 b8  04 00 00 00     #moving print
       out syscall number to eax
13 cd  80  #calling the linux kernel to execute our print
       to stdout
14 cd  80  #calling the linux kernel to execute our print
       to stdout
15
16 b8  01 00 00 00 #moving sys_exit call number to eax
17 cd  80          #executing it via linux sys\_call
18 cd  80          #executing it via linux sys_call
```

Figure 3.1: Printing Hello, World in Machine Code *Thanks to cedriczirtacic on github for the code*

Assembly Languages
Assembly language is a low-level programming language in which the instructions in the language correspond to the architecture's machine code instructions.

While a step above machine code, Assembly, which is still considered a low-level language, requires the use of an assembler; a platform dependent program that converts the program to machine code. Each statement would still be a direct action for the computer. This might include allocating memory, accessing variables, and performing calculations. The syntax of the commands are often cumbersome and confusing.

As an example, the code in figure 3.2 would be used to print *Hello World* on an x86_64 processor running Linux.

```
1    .global main     /* Make main a global function */
2 main:           /* Start function main */
3    mov $msg, %rdi   /* RDI gets pointer to message */
4    call puts        /* puts(msg) */
5    mov $0, %rax     /* RAX gets zero */
6    ret           /* return(0) */
7 msg:            /* Declare a label for the string */
8    .asciz Hello World  /* Define the string */
```

Figure 3.2: Printing Hello World in Assembly Language *Thanks to Jack Brennen of Google*

Programs written in a low-level language run directly on the computer's processor. This, inherently, will make the program fast to run. But the difficulties may far outweigh the speed.

The challenge in writing a program using a low-level language is obvious from the two examples. Part of this complexity is a result of the need for the program to direct the computer to perform every action - nothing in the code is directly interpreted by the computer. In effect, the program must assign each item to an appropriate place in memory and then perform the actions using the memory locations. Each time a new variable is created memory must be allocated for the variable. When the variable is deleted memory must be freed up, or deallocated.

But the issues of low-level languages are more than just memory management. For most programmers the problem is that there is no simple, easily understood, command that can be used to direct the computer.

Abstraction
Abstraction is the process in which actions are derived from their context.

It is this aspect of the low-level language that is being described when it is said that low-level languages lack *abstraction* - no operations can be inferred by the command.

To address the challenges of programming in machine code, and allow for portability of the software, programmers created *high-level languages*.

High-Level Languages

Most modern programs are written in one of the many different high-level languages. The original goal was to create a language for writing programs that was more similar to the syntax of a spoken language.

High-Level Languages
A programming language with strong abstraction.

With the simplicity of a readable programming language, high level languages have strong abstraction. This means that

many of the details of making the computer operate are already built into the language.

As an example recall the low-level assembly code in figure 3.2 that was needed to print the phrase *Hello World*. Now let's write the same code but in MatLab.

```
1 fprintf('Hello World\n');      % Prints the string
```

Figure 3.3: Printing Hello World in MatLab

The code in figure 3.3 is still a bit cryptic. Why **fprintf** and not just **print**? Why the **\n**? These items will be addressed later but for now if you showed a non-programer the line of MatLab code and asked them what it did they would probably say it prints the phrase *Hello World*.

The difference between the two is that in the Assembly code there was no abstraction - we needed to explicitly tell the computer how to read the letters of the phrase into memory, where to store them, and how they could be printed.

But in the high level language there is strong abstraction. All that we have to do is say print the phrase and the programming language takes care of all of the other details. The computer still has no direction on how to do any of these actions, it is just that the high level language has taken care of it for us.

3.2 Types of Programming Languages

There are two characteristics that are used to identify modern programming languages; how they work, and how the programmer implements them. As for how the language functions, a language can be procedural or object oriented. This is known as the programming model.

The second characteristic addresses how the program is written or implemented. With respect to how the programmer writes the program, the programming language can be compiled or interpreted.

3.2.1 Programming Model

The programming model describes how the program operates. The two primary models are *object-oriented programming*, and *procedural programming*.

Object-oriented programming is based upon the creation of classes and objects. A class is a structure consisting of both variables and methods - smaller program instructions - whereas an object is an instance of a class. Thus there can be many different objects that are all of the same class. The objects then interact with both each other and with data streams outside of the program.

Object-oriented programs can be thought of as modular. Each aspect of the program is a separate module and the modules interact with each other.

While object-oriented programming has been around since the 1950s, it has only become a commonly used programming paradigm in the last twenty years when it has surpassed the use of procedural programming.

Procedural programming is the original programming language model. In this approach the programming creates separate procedures, or functions, or methods. These functions are often compared to mathematical functions in that the user provides a set of inputs to the function. The function then performs a set of operations on the input data. The function may then perform an output action such as printing or writing to a file, or more commonly it will return - or send - a result back to the function that initiated the original action to the current function.

A goal of procedural programming to implement *top-down design*. This is when the program is broken down into a set of tasks. Each of these individual tasks are then broken down into subtasks. This process is repeated until each task that needs to be done is trivial. At this point, the trivial tasks are written as separate functions. The program then consists of a *driver* function that calls other functions to complete tasks as they are needed.

The programming model describes how the language operates in solving a task, but there are also differences in how certain languages are written and run. Languages can be *compiled* or *interpreted*. This describes how the program is translated from a text into something the can be run - or executed - on a computer.

3.2.2 Program Translation

Translating a program is the process of converting the plain text that was written by the programmer into a form that can

be run on the computer. There are two common types of program translation - compiling or interpreting.

Compiled Programming Languages
A program that is written in a *compiled* language will start with one or more *source code* files. This is a text file with the individual program commands. While every language is a bit different, the source code file must be written as a complete program. If the language requires a specific start command and an end command, the source code must include these.

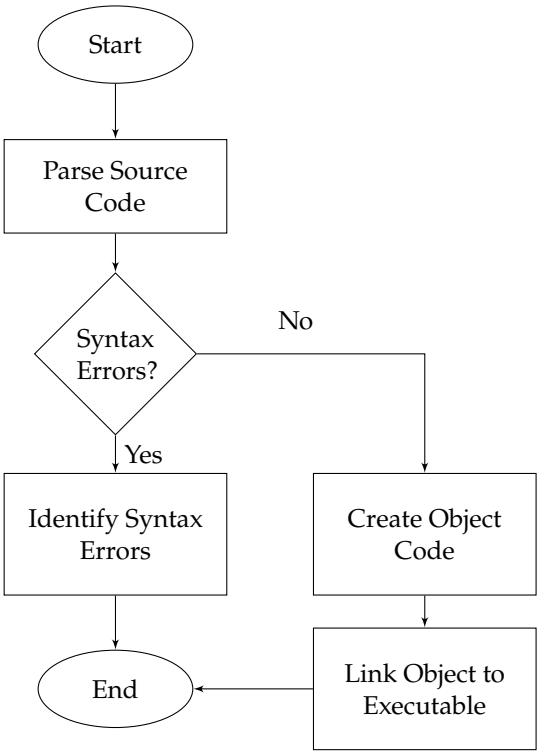


Figure 3.4: Compiling a Program

When the source code is in a complete form it can be run through a *compliler*. The compiler is a platform dependent program that is will convert the source code into an executable file. More specifically, the compiler is translator in that it translates the high level language of the source code into the machine code needed to run the program on the computer.

The compiler first strips all non-executable content from the file. This includes any comments or white space. It then *parses* the file. Parsing is the process of reading the file into memory one character at a time while checking it for correct syntax. If

an error is found the compiling process stops and usually error messages are sent back to the programmer.

If no syntax errors are found, the program is converted to *object code*. Object code is a file where the program has been converted into binary machine code. The object code is then *linked* - that is the compiler uses the object code to create an *executable* file. The executable is the program file; it can be run by the computer. It is *platform dependent* meaning that it can only be run on the same type of computer on which it was compiled. A powerful aspect of the executable is that once it is created it can be run on multiple computers - as long as the computer is running the same operating system - without the need for the computers to have the compiler, or the source code.

The process of compiling can be time consuming. It requires the program to be in a complete state. This might be mean only that is has a start and end. The entire program is parsed and linked. If there is a single error in the program the process stops without a single line of code being run. Once the syntax error has been identified and corrected, the compiling process can be started again.

Compiled Programs

The paradigm of compiled computer programs is that they are *Slow to Code, but fast to run*

While modern compilers are fast, compiling a long program can be time consuming. The upside side is the the executable program is does not need to be checked for any errors. Further, it is usually optimized by the compiler to run efficiently on the machine on which it is written. This results in a common saying about coding in a compiled programming language - *Slow to code, but fast to run*.

The use of compiled programming languages had become the standard because of the speed of execution when the program is run. But there is an alternative, and while the programs run more slowly than compiled languages, with today's fast processors they have made a resurgence. These are *interpreted programming languages*.

Interpreted Programming Languages

The alternative writing a program in a compiled programming language is to use an interpreted programming language. While compiled programs and interpreted programs may appear the same to the user, to the programmer they are very different.

The first difference in an interpreted program is semantic. The compiled program was written as source code. But the

interpreted programming languages use a *script*. The reason is how they function. The script can be compared to a script that might be used in a play or a movie.

Run Lines

In the theatre, when an actor is practicing the play with only one other person it is said they *run lines*. Similarly, the computer will *run lines* when it executes the script.

In a play script each line is either a dramatic action - something said - or a physical action - something done. While the actor performs that action the other actors are theoretically idle - they wait. This is similar to how the script is run.

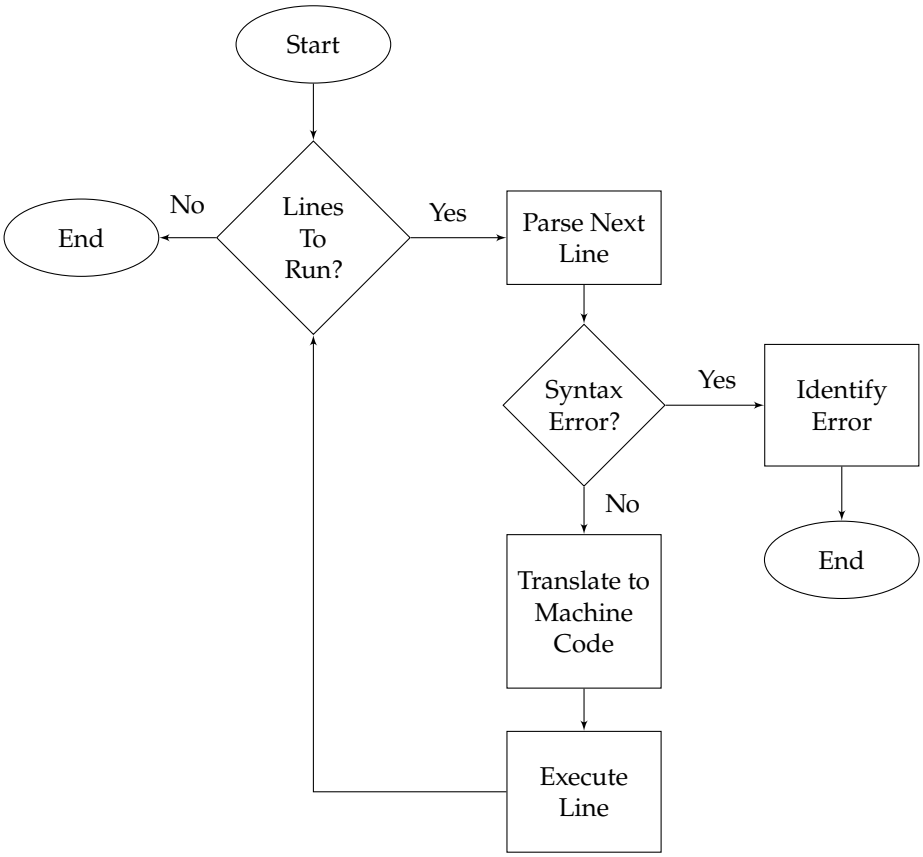


Figure 3.5: Interpreting a Program

Programs written in an interpreted programming language act on a single line at a time. To do this, the program requires an interpreter to translate the line of code to machine code to be run.

The interpreter is a program written for a specific operating system. In its simplest form, It opens the computer script and runs the program. More specifically, the interpreter reads the script into memory. It then parses the first line of the program.

If it finds a syntax error, the interpreter stops the program and prints an error message. But if it does not find a syntax error the interpreter translates the single line of the script into machine language and runs the line. If there are more lines of code in the script, the interpreter goes to the next line and repeats the process. This continues until it either finds an error or there are no more lines of code in the script.

Interpreted Programs

The paradigm of interpreted computer programs is that they are *Fast to Code, but slow to run*

One way to view interpreted programs is that the parsing and translating are performed on each line of the program every time the script is run. There is no executable created but instead the script is the program. Historically this meant that the program would run more slowly than would a compiled program. But there was an upside. The interpreter only needed a single line of code to run. The process of writing a script becomes incremental. You write a line and run the program. If it works, you write another line of code, building the program one line at time. Each time the interpreter is run, all the previous code has been checked, and while it is parsed a second time, you already know that it is correct and will not need to be updated. This makes the process of coding faster than with a compiled language. Thus fast to code, but slow to run.

Note

The idea that the interpreted programs are slow to run has become less of an issue with the faster processors in modern computers. While they still run more slowly than do compiled programs, the differences are much less noticeable.

Another difference between compiled and interpreted programs is with respect to platform independence. Recall that a compiled program is optimized to run on a single platform. Thus you cannot run an executable on any computer. But the script for an interpreted program is plain text and can be run on any computer that has the proper interpreter installed. Thus interpreted programs are considered to be platform independent.

3.3 Summary

An algorithm is the process we follow to complete a task. To implement the algorithm we write a program. But to get the program to run on a computer requires a specific syntax, and thus a specific programming language.

While mechanical computers would be programmed by physically manipulating gears or levers, programs for electronic computers consist of a set instructions that the computer can follow.

In its most basic form - the low level language - these instructions are directly readable by the computer, for example machine language, and control the computer's central processing unit. In this type of language every step must be addressed explicitly. The difficulty in this type of programming becomes obvious for all but the most trivial programs.

High level languages have been designed to make the programming process more accessible. The goal is enable the programmer to enter the commands in a syntax similar that of a spoken language. If the program wants to add two values, the program tells them to be added. To print the result, the program would use a command that simply tells the computer to print.

While the idea of the many different high level languages is the same, the means of writing the program varies widely. With respect the language model, they can be object-oriented or procedural. As how the computer runs - or translates - the program, there are compiled programming languages and interpreted languages.

Object-oriented programming languages allow the programmer to create modules, sometimes described as small programs, that interact with each other or with devices outside of the computer. In a procedural programming language, the programming creates a separate routines, or functions or methods, to perform individual tasks.

Whether object-oriented or procedural, the program still needs to be translated into machine code to be run. Some languages do this by compiling a source code while other interpret a script.

Compiled programming languages will parse the entire source code, identifying any syntax errors that might exist. If there are none, the compiler then translates the plain text source code file into object code. The object code is then linked to an executable file. The executable is binary machine code that has been optimized that can be run directly on the computer without further processing. The executable is *platform dependent* meaning that it can only be run on the same type of computer on which it was compiled.

The process of compiling a program often makes the programming process time intensive. But once compiled the executable is efficient. Thus the paradigm of compiled languages as *slow to code, but fast to run*.

Interpreted programming languages require an interpreter that is run on the computer. The interpreter reads the plain text script - the program - into memory. It then parses a single line, and if there are no errors it converts the line of the script to machine code and runs it. It then goes on to the next line and the next until it either finds an error or finishes the program.

The process or parsing a single line and then running it makes the programming process a much faster endeavor. The programmer can write a single line of code, test it, and if it is correct add the next and the next. This can greatly speed up the programming process. The tradeoff is in that the each line must run individually. Thus interpreted languages are said to *fast to code, but slow to run*.

While the interpreter must be written for a specific type of computer, the script does not. Since interpreters are available for most of the common computer types, a script written on one computer can usually be run on any other computer regardless of whether it is the same type. Thus interpreted languages are considered the *platform independent*.

MatLab is an interpreted programming language, and while the programs can be written as an object-oriented program, the programs can also be written as a procedural language.

Starting with Programming 4

To err is human. To really foul things up requires a computer.
Bill Vaughan - Journalist

If our only interest was computing it would not be necessary to ever turn on a computer. We could study every aspect of computing without ever writing a program. But what fun is that?

Code
The common term for writing computer programs is coding, as in *she is coding the new interface.* The program itself is often referred to as the *code*.

The truth is that we learn computing in order to understand the the theory that we need to write effective computer programs. Our goal is the program - or *code* as it is often called. It is through the program that we are able to get the computer to perform tasks for us.

MatLab is an interpreted programming language. As with many interpreted programming languages MatLab can be run interactively or through a written script.

4.1 Interactive Programming

Interactively
Running a program *interactively* consists of entering a single statement directly into the command line.

Because an interpreted programming language is executed one line at a time, many interpreted programs can be run *interactively*; that is by entering a single statement on the command line. MatLab is one of those programming languages.

A simple example of interactive programming is to use MatLab as a calculator

In each case in figure!4.1 the operations are entered on the commend line as a simple arithmetic operation. When you press enter on the keyboard the single line is executed - in this case performing the arithmetic operation,

4.1.1 Arithmetic Operations

MatLab has operators - the symbol that indicates that the program should do something - to perform the four standard mathematic operations and a fifth for exponentiation - raising a value to a power.

```
>>  3 + 5

ans =

    8

>>  9 - 4

ans =

    5

>>  2 .* 5

ans =

    10

>>  36 ./ 9

ans =

    4

>>  3 .^ 4

ans =

    81

>>
```

Figure 4.1: Interactive arithmetic

Table 4.1: Arithmetic Operators

Operator	Action	Example
+	Addition	a + b
-	Subtraction	a - b
.*	Multiplication	a .* b
./	Division	a ./ b
.^	Exponentiation	a .^b

Each operator does in a program exactly what it would do on pencil paper.

Dot Operator

While we are all familiar with these operators there is an unusual component in the product, quotient, and power operators - a dot or period in front of the operator. It is often called the *dot operator* because of this.

Warning
The difference between using the dot and omitting it is an important consideration when the two values are vectors or matrices. With the dot the operations are performed *element-wise* while without

The dot operator indicates that the operation is to be performed *element-wise* instead of being done using matrix operations. At this point we are only performing operations on scalars and while the dot operator is the correct syntax you would still get the correct result if you had not included it. But since the operations are intended as element-wise it is important to include the dot.

Order of Operations

An issue can arise when the order in which the arithmetic operations is ambiguous. Is $3 + 5. * 4 = 32$ or is it 23? It becomes clear when parentheses are added; $(3 + 5). * 4 = 32$ while $3 + (5. * 4) = 23$.

But what if the parentheses are omitted? With or without the parentheses arithmetic operations are performed in accordance with the standard algebraic order of operations; *PEMDAS*.

Order	Description	Operator
1	**P**arentheses	$(\cdots)$
2	**E**xponentiation	$.\wedge$
3	**M**ultiplication	$.*$
4	**D**ivision	$./$
5	**A**ddition	$+$
6	**S**ubtraction	$-$

Table 4.2: Algebraic Order of Operations

Note
The order of operations is easy to remember with any of a number of mnemonics. As a single word, *Pemdas* or as a quick phrase such as *Please excuse my dear Aunt Sally* or a variation provided by a student *Please excuse my dumb ass sister*.

The order of operations show that all operations that are enclosed with parentheses take precedence. As an example

$$\begin{aligned}(3 + 5) .* 7 &= 8. * 7 \\ &= 56\end{aligned}$$

while

$$\begin{aligned}3 + (5 .* 7) &= 3 + 35 \\ &= 38\end{aligned}$$

After this it is exponentiation or raising to a power.

$$\begin{aligned}2 .\wedge 3 + 4 &= 8 + 4 \\ &= 12\end{aligned}$$

but

$$\begin{aligned} 2 .\wedge (3 + 4) &= 2 .\wedge 7 \\ &= 128 \end{aligned}$$

4.1.2 Variables

Variable
A variable is a block of allocated memory to which data can written and read. It is most commonly used in a program as temporary data storage.

Notice that in figure 4.1 each time that you enter a value or operation on the command line the system responds with the result but also an **ans =**. This would be same for any operation or even a single variable.

```
>>  7

ans =

    7

>>
```

Figure 4.2: Storing results in the variable **ans**

In fact you could enter a number on the command line, as in figure 4.2, and the system would respond with the same **ans =**. This is an example of *assigning a value to a variable*.

A *variable* is a block of memory that has been allocated to store data. Each memory block has an address that can be used to identify the location of the memory.

Identifier
In programming an *identifier* is a name that is given to some item so that item can be accessed in the program. A common identifier is the variable identifier - a name given to a variable in the program.

While each block of memory has an address assigned to it, keeping track of the constantly changing memory addresses would be a herculean task. Instead, the variables have variable identifiers - common names that can be used instead of the variable address.

In the sample code in figure 4.2 *ans* is a variable identifier - although from here on the variable identifier will be called simply the variable. Because *ans*directs the program to a location in memory, data can be written to that location or read from it by using the variable *ans*.

Comment
The terms *variable* and *variable identifier* indicate the memory location and the name that the programmer gave the variable. This formality is never enforced. Any time that the term *variable* is used it is implied that you mean the variable identifier. Further, if it is the memory address that is needed the term *variable reference* or *variable address* would be used.

Changes to the variable are made using the = called the *assignment operator*. The action is called an *assignment* or *assigning data* to the variable.

When using the assignment operator any operations must be performed on the right hand side of the operator. The program will perform those operations first and the assign the results to the single variable - if any - on the left hand side

of the operator. If there is no assignment operator then the results are stored in the variable *ans* by default. If *ans* already exists the data will be overwritten.

```
>>  ans = 5 + 4

ans =

    9

>>
```

Figure 4.3: Storing results in the variable **ans**

The assignment operator should not be confused with a mathematical equals sign. It does not indicate equality. In fact in a program the assignment operations would rarely make sense in a mathematical form. An example is having the variable on both sides of the assignment operator.

Assignment Operator
The = is called the *assignment operator*. It directs data to - or from - a variable.

A common action is to have the variable on both sides of the assignment operator. This is known as an *update*. Because the program will always perform all of the actions on the right hand side before any assignments, the current value of the variable will be used in the operations. Once completed the results are then copied into, and thus replacing the old values in, the memory location.

Warning
Although the assignment operator appears like a mathematical equals sign it is important to understand that it is not an indication of equality.

```
>>  ans = 9

ans =

    9

>> ans = ans ./ 2

ans =

    4.5

>>
```

Figure 4.4: Updating the variable **ans**

Creating Variables

If you do not create a variable in your command, MatLab will create one for you. In Matlab this default variable is always *ans* (a diminutive of answer). It is limiting to have only a single variable so you can create others as needed.

```
>>  x = 16

x =

    16

>> y = 3.* x - 5

y =

    43

>>
```

Figure 4.5: Creating additional variables

There are no limits on the number of variables that you may create in a program, and only a few limitations on what you can name them. The rules are quite simple.

- A variable identifier must begin with a letter of the Latin alphabet
- After the first letter the variable may contain any alpha-numeric character or the underscore symbol, but no other symbols or spaces
- Variables are case sensitive thus a variable with an upper case letter and the same identifier but with lower case letters are two different variables. There are no restrictions on where an upper case or lower case letter may be used
- A variable may not be a keyword. You cannot use names such as **function**, or **end**, or **while**, or any of the additional reserved words.

Variable Nomenclature

- A variable identifier must begin with a letter of alphabet
- All other characters may be any alpha-numeric character or the underscore symbol
- Variables are case sensitive
- No keywords

Keywords are words that are reserved for specific programming purposes. Some common examples are the words *function*, *end*, *if*, *while*, and *for* but there are far too many to list here.

No worries, MatLab provides a means of determining if a possible variable name is a keyword. It is the command **iskeyword**.

In the sample code in figure 4.6 the word *logical* that appears indicates that the result will be one of the values 1 or 0. This is known as a logical - an output that indicates if the test was true or false. The logical value 1 indicates **true**, while the logical value 0 indicates **false**.

Logical
A logical is a variable that is used to denote that a test is either **true** - the value **1**, or false - the value 0

There is another function, **isvarname**, figure 4.7, that can be used to check if your variable name is legal. Much like

```
>>  iskeyword end

ans =

   logical

    1

>>  iskeyword time2go

ans =

   logical

    0

>>
```

Figure 4.6: Checking variable identifiers with **iskeyword**

iskeyword this function returns the logical **true** - or 1 - the name that you used is legal, and a logical **false** - or 0 - if it is not.

```
>>  isvarname 3xy

ans =

   logical

    0

>>  isvarname xy3

ans =

   logical

    1

>>
```

Figure 4.7: Checking variable identifiers with **isvarname**

Data Types

Data Type
Each value in a program has a *data type*. It is an attribute of the value that is used to determine how the program is to use the data. The three primary data types are *integers*, *floating point values*, and *characters*.

MatLab is what is known as a weakly typed language. This has nothing to do with keyboards but instead is a description of how variables deal with data of different *data types*.

The data type is an attribute of the data that is to be stored. There are three primary data types; integers, floating point

values, and characters. Since MatLab is a weakly typed language it allows a variable to store data of each of these, and in addition, strings.

If the data is value then MatLab will determine if it is an integer based upon its context. That is if the value contains a fractional part then it will assume that it is a floating point value. If it does not then it will assume that the value is an integer.

A issue might occur with characters or strings. If you want to store a character or string in a variable by simply writing the character or string the program will assume that it is another variable. If the character is a symbol then the program would assume that it is an operation. As a result of this all characters, strings, and symbols are placed within a pair of single quotes.

```
1 >>  name = 'Joe Bfstk'
2
3 name =
4
5 'Joe Bfstk'
6
7 >>
```

Figure 4.8: Storing a string in a variable

The interactive approach is nice if all you need are some simple results; calculating a few values, plotting the graph of a function to print, or finding a single solution to a small system of linear equations.

But for almost anything else running your program from the command line is horribly inefficient. You need to enter each line of a program individually, and if you want to make changes and then run it again you have to start over. Imagine the frustration of entering fifty or a hundred lines only to make a numerical mistake on line one hundred and one.

To counter this you need a means of running a program that is consistent and repeatable. You want to create a script.

4.2 Scripted Programming

The interactive approach to programming works, but lacks efficiency and repeatability. The chances are good that once you write a program you would want to run it multiple times - changing a parameter or two and calculating new results. The would require a program in which all of the executable

statements can be run multiple times with little or no changes to the program.

Since MatLab is an interpreted language this requires writing a *script*.

Script
A *script* is the program that is written in an interpreted programming language. It consists of a set of commands that will be executed in the sequence in which they are presented.

Recall that a script is the set of executable commands that will be followed in a specific order. An interpreted language requires an interpreter that for each line of code in the script

- parses the line of code
- checks the line for syntax errors
- if an error is found - exits the program
- else
  - translates the line to machine code
  - executes the statement
  - goes to the next line

This process is similar to an actor reading a play. The actor reads a line or performs an action. Until that action is completed - and completed successfully - the rest of the play waits. The directions to the actors is called a *script* thus the program that is written for an interpreted programming language is also called a *script*.

4.2.1 Writing a script

In its simplest form, a script is simply a text file in which the lines of code are written in the order in which you want them to be run. We are going to take a more formal approach to programming that will be better explained in chapter 5.11 but for now just do it.

The file consists a line with the name of the program - which will call **driver**, the statements that we want the program to run, and the keyword **end** to indicate that end - obviously - of the program.

In MatLab this file is often called a *dot m* file because of the **.m** suffix that you will give the file. It must have the same name as the program name - in this case **driver** - so the file will be named **driver.m**

Tradition has it that the first program that you write will print the statement *Hello World!* on the screen. So let's do it

Once you have written the script, you run it by typing the name of the file onto the command line. It is a standard convention to include a set of parentheses, but for now that is

Story
Why do we always start with *Hello World!*? You could start by doing any programming task but printing an output is always first.
There is a puzzler that goes "If you want to teach a dog to swim, what do you do first?" Ask a hundred people and you will get a hundred answers - but rarely are any of them the correct one. The answer is "you teach them to get out of the water."
The first skill is not to get into the water, it is to get out. If the poor dog cannot get out of the water then all of the swimming lessons will be for naught.
In the same way the first programming skill that you need to learn is how to get the information out of the program. Without that you would never know if the program was performing the correct actions or even any actions - thus *Hello World!*

optional. The program would run the same with or without them.

```
function driver()

    s = Hello World!

end
```

```
>> driver()
s = Hello World!
>>
```

Figure 4.9: The Hello World Program

This first program was simple but it demonstrates the ideas of a script. The single line of code is parsed - read into memory one character at a time while checking for syntax errors, and then executed.

In this program, a string of text was assigned to a variable and because the line does not end with a semi-colon the value stored in the variable is printed to the screen. Echoing is not the method we want to use for printing but for now it accomplishes the task.

4.2.2 The form of a program

As we begin to write programs - what we will call *scripts* - it is helpful to develop an efficient approach. Too often programmers will start with a blank screen and starting writing inputs, then calculations, then outputs. After all, this is the form the program will take when it is done.

But it is not how it should be written. While it is the final form of the program, it is inefficient. Instead, a more productive approach is to work in reverse. Create the outputs with all of the formatting to have the results in a form that can be either printed or can be used directly. To do this you will need variables with values. This is what we do first. Do not be concerned with the calculations - they will come later.

Variables in a program can be classified as input variables or output variables. The values in the input variable will later be entered by the user of the program, while the output variables will be the result of calculations. Regardless of the type, at this point they should just be assigned values. This is often called *hard coding*.

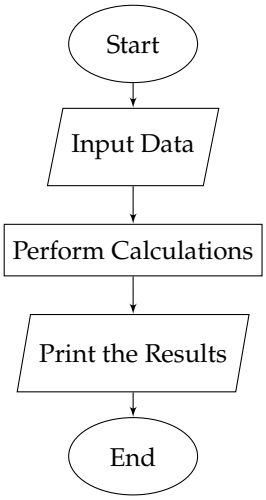


Figure 4.10: Standard form of a computer program

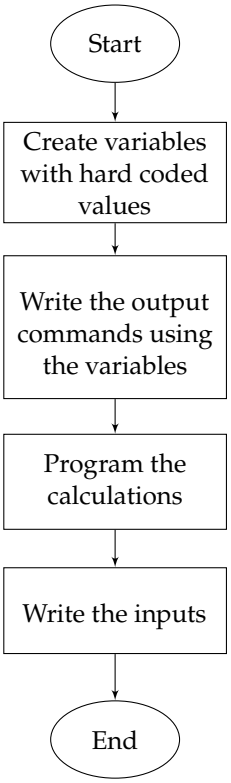


Figure 4.11: Process of efficiently writing a program

Once the variables in the program have been created and assigned values - both those that will be the inputs and those that will be the outputs to the program - start writing the outputs.

4.3 Creating Output

A paradox of programming is that we need to be able to print before we learn to enter data, or perform calculations. But why?

While the program we wrote could have performed any task, how would we know it was correct - or even if it ran at all - if we do not have an output? The output is where the user gets to see the result of the program. So while printing the results seems like it should be the last task, it is actually the first.

In the *Hello World!* program you had the script print by calling a function **fprintf**. There are actually several different ways to print results in your program. Each of them have their own purpose and should be used as such.

4.3.1 Echoing

Echoing
Echoing is when the results of an operation is printed to the screen as soon as the action is complete. It is activated by *not* ending a line with a semi-colon (**;**).

The first means of printing is called *echoing*. Echoing is when the results of an operation, be it an input, assignment, or a calculation are printed to the screen immediately after it is done.

Flag
A flag is a variable or indicator of a state in the program. It can only have two values - On or Off, 1 or 0, true or false, and so on.

Echoing is a type of *flag*. In programming a flag is a state in the program that indicates that some condition has been met or that some operation should be done. It is called a flag because it only has two possible choices; On or Off - similar to a flag being raised or lowered.

To use echoing you omit the semi-colon from the end of a line. Examples of this are shown in the sample code in figures 4.3, 4.4, and 4.5.

When the semi-colon is at the end of the line the statement is run exactly like it was before, but nothing is shown on the screen.

Warning
Echoing is a useful technique for checking intermediate results or when using the command line for quick calculations. But it should not be used as the main method for printing results in your programs.

Echoing should not be used as the standard print method. Instead it is a *debugging* tool. When you run your program you can turn echoing on (by removing the ;) to check if intermediate results are correct.

But if you use it throughout you program it becomes a nuisance. Values - in which you are not normally interested - continue to clutter the output. They become a distraction. Instead, once you are assured that the intermediate calculations are correct it is important to turn echoing back off (by ending the line with the ;).

Since echoing should only be used for debugging, standard practice is to end every line of code with the semi-colon; even those that would have printed without the semi-colon.

4.3.2 Display

disp
The **disp** function will print strings of text or the values stored in a variable.

There is another print function that while it has a use for vectors and matrices is not recommended for normal printing. This is the **disp** function

```
disp(variable);
```

Figure 4.12: Syntax of the **disp** function

Unlike echoing in which the variable identifier and the value are printed the **disp** function will print the variable's data and nothing else. This function can be used print both text - or *strings* - as well as numerical data.

```
function driver()

    x = 7 ./ 6;
    disp('Printing using disp');
    disp(x);

end
```

```
>> driver()
Printing using disp
1.667

>>
```

Figure 4.13: Printing using disp

The default for **disp** is to print the string or value exactly as presented with no other formatting or alignments. It does not print the variable identifier.

The default is to print values using with four digits after the radix point. If the number is small enough it prints it directly while for larger - or smaller values **disp** prints using scientific notation.

It may occur that you need to know more than four digits in the fractional part. If so you will need to adjust the precision of the output (the number of decimal places).

There are several system commands, table 4.3, that can set the precision that are printed using **disp**. Since these are system commands once they called they fix the formatting for echoing and for any other system printing until they are changed or the MatLab session is closed.

Table 4.3: Setting precision for the **disp** function

Format Command	Print
format bank	Two digits after the radix point
format compact	Four digits after the radix point
format long	Fifteen digits after the radix point
format shortEng	Engineering format - scientific notation limited to multiples of three - with four digits after the radix point
format longEng	Engineering format - scientific notation limited to multiples of three - with fifteen digits after the radix point
format shortG	Either fixed decimal format or scientific notation whichever is shorter each value with four digits after the radix point
format longG	Either fixed decimal format or scientific notation whichever is shorter each value with fifteen digits after the radix point
format hex	Gexadecimal (base 16)
format rat	Ratio of small integers
format +	+ for positive values, - for negative values, and blank for zero
format	Reset to the system default

Warning
The **format** command is a system command. This means that it does not apply only to **disp** or to that run of the program. It applies to both echoing and **disp** and will remain until it is either changed or the MatLab session ends.

Printing is a primary task in most programs. And while you can print using echoing, it is intended as a debugging tool and should not be used in the final program.

Further **disp**, while providing a basic means of printing text and data does not include many of the formatting tools that are necessary for controlling how the output is printed.

```
function driver()

    disp('Using default format');
    x = 53287 ./ 3;
    disp(x);
    format shortEng
    disp('and now with engineering notation');
    disp(x);

end
```

```
>> driver()
Using default format
1.7762e+04
and now with engineering notation
17.7623e+003

>>
```

Figure 4.14: Changing the precision using the **format** command

So while we will continue to use echoing - as a temporary means for printing - **disp** has been superseded by a printing function that offers more ability to control almost every aspect of printing. This is the function **fprintf**.

Why is the function fprintf and not just print?
The first f stands for file while the second is for formatted. You use this same function for writing results to files thus the first f. The second part is also important. Formatting means the we control the width of the printed value (the number of spaces that is uses) and also the precision (the number of decimal places).

4.3.3 fprintf

A function in the old C programming language provided a method of printing both strings and data simultaneously. In addition the function enabled a wide range of formatting. This function has been adapted for MatLab as **fprintf**

```
fprintf('formatting string', ouput parameter list)
    ;
```

Figure 4.15: Syntax of the **fprintf** function

The formatting string can be as simple as a string of text to be printed. It can - and usually does - include indications as to where to print data. In the sample code, figure 4.9, since all that you were going to print is *Hello World!* the formatting string contained just that. The single quotes at the beginning and end are used to indicate the start and end of the formatting string.

We want more than being able to print strings of text. We want to print our results. This will require formatting.

Formatting the Output

How would you print variables? If you could type them in as text then you just put them inside of the quotes but that is a bit paradoxical. To print them you need to know the values when you write the code. But if you already know the values then there is probably no reason to write a program.

Newline
There are a pair of characters right before the closing quote. The \n is called an *escape sequence*. It tells the program to go to a new line at the end of the printing.

You want to be able to print the values after they have been calculated. This is going to require formatting. We can add several different types of variables to printed. The most common are strings of text, and numerical values.

The location in which you will print text is indicated with a %s while numerical values are positioned using one of several formatting characters. The most common are **%f**, **%d**, **%e**, and **%g**.

```
1  function driver()
2
3     name = 'Joe Bfstk';
4     month = 'August';
5     day = 23;
6     year = 2025;
7     fprintf('Hello %s!\n', name);
8     fprintf('It is %d %s %d\n', day, month, year);
9
10 end
```

```
1  >> driver()
2  Hello Joe Bfstk!
3  It is 23 August 2025
4
5  >>
```

Figure 4.16: Printing your name and date in which name is a string of text and the date consists of two integer values and a string

In the sample code in figure 4.16 the location of the formatting characters indicate where the data will be printed. Which variable is printed is determined by the order that they are presented - first in the list gets printed first, second is second, and so on.

Controlling the width and precision

Precision
Precision is distance of an observation from the actual value. In the case of coding, precision is determined by the number of decimal places that are retained or printed.

In the standard form the formatting characters provide the location for printing the data value. When they print, whatever is printed abuts the text that comes both immediately before an after it. Additionally, if what is being

Table 4.4: Formatting Characters

Format Character	Action
%f	Print a floating point value
%d	Print an integer value
%e	Print a floating point value in scientific notation
%g	Print a floating point value in either regular notation or scientific notation whichever is shorter
%o	Print an integer in octal (base 8)
%x	Print an integer in hexadecimal (base 16)
%c	Print a single character
%s	Print a string of text

printed is a a floating point value - a number with a fractional part (or decimal places) - then the default is to print the value with four decimal places.

You may want additional control over the printing. Perhaps you want to align several rows of values o the radix point (the decimal point in base ten). **fprintf** also provides the ability to control the amount of space - the width - that is reserved for the printed data, and in the case of printing floating point values, the precision.

Radix Point
The radix point the period that is used to separate the integer part of a floating point number from its fractional part. In base ten it is commonly called the *decimal point*

Each of the formatting characters allow you to set the width that this reserved for printing. This is done by adding a non-negative value between the % and the letter. For example, to print an integer right justified with a minimum of seven spaces reserved you would use **%7d**. This applies for any of the non-floating point formatting characters.

The syntax for doing this with an integer variable is shown in figure 4.17. Of course the *w* would be replaced with an integer value.

```
1 fprintf('%wd', integer_variable);
```

Figure 4.17: Syntax for reserving *w* spaces for an integer

This same addition will work with all of the non-floating point data types. For example, **%32s** will reserve 32 spaces for a string of text.

```
function driver()

   name = 'Joe Bfstk';
   month = 'August';
   day = 23;
   year = 2025;
   fprintf('Hello %s!\n', name);
   fprintf('It is %d %s %d\n', day, month, year);

end
```

```
>> driver()
Hello Joe Bfstk!
It is 23 August 2025

>>
```

Figure 4.18: Printing your name and date in which name is a string of text and the date consists of two integer values and a string

A common method to attempt to align text is to add spaces within the string in the formatting of the **fprintf** function. It may work but is awkward. It also tends to fail when the size of the values being printed change from run to run.

Alternatively, the width parameter when printing strings of text can be a useful technique for aligning columns in a table. An example of this is in figure 4.18

```
function driver()

   fprintf('\n');
   fprintf('%15s%20s%20s\n', 'Trial', 'Voltage', '
    Power');
   fprintf('\n');

end
```

```
>> driver()

     Trial          Voltage          Power

>>
```

Figure 4.19: Printing a table header using string literals

In this example, the three strings on the right of the comma are actually variables, but are hard coded into the **fprintf** function parameters as literals. When the program is run, the literal is printed in the space that is reserved for it with the %s. You could have also assigned the text to a variable and

used the variable in the fprintf statement as was done in figure 4.19.

Most of the variables printed are not integers or text, but are floating point values. Recall this means the value has a fractional part; what we usually think of as a decimals or values to the right of the radix (the decimal point).

Note
When printing a floating point value, the printed value is rounded using the standard rounding rules.

To format a floating point value you enter the width and the precision separated by a period.

```
1 fprintf('%w.pf', floating_variable);
```

Figure 4.20: Syntax for reserving a total of w spaces and p decimal places for for a floating point value

In this method, the value in w is the total number of spaces reserved for the value. This includes the decimals. p is the precision - or the number of decimal places.

```
1  function driver()
2  %   DRIVER driver() is the main or driver function
3
4      %   Assign data to all variables
5      trial1 = 42;
6      volt1 = 109.74652;
7      power1 = 203.93287;
8      trial2 = 43;
9      volt2 = 73.29876;
10     power2 = 12.30842;
11
12     %   Print the values as a formatted table
13     fprintf('\n');
14     fprintf('%15s%20s%20s\n', 'Trial', 'Voltage', 'Power');
15     fprintf('%15d%20.2f%20.2f\n', trials, volt, power);
16     fprintf('\n');
17
18 end
```

```
1 >> driver()
2
3      Trial          Voltage          Power
4       42             109.75          203.93
5       43              73.30           12.31
6
7 >>
```

Figure 4.21: Printing strings, decimals, and floating point values

By adjusting the width, it is possible to get the values in each printed line to align on the decimal points.

Escape Sequences

There is an additional item of text in the **fprintf** formatting string. It is the \n, what we earlier described as the command to force a new line when printing. This is known as an *escape sequence*.

Table 4.5: EscapeSequences

Format Character	Action
\n	Force a new line
\r	Carriage return
\t	Tab over the print to the next stop
\b	Move back a space
’’	Print a single quote - ’
""	Print a double quote - ”
%%	Print the percentage symbol - %

Escape sequences address the issue of printing a character that if you had entered the actual keystroke would have been interpreted differently. For example, the *enter* or *return* key on the keyboard moves the cursor to the next line. When you press it while writing a program, the program would move to the next line. This would not put the new line in the print statement, but would actually result in a syntax error since the formatting string would have stopped before the closing quote.

Figure 4.22: The author’s manual typewriter that he took to college in 1979

To resolve this, programs use escape sequences for the different non-enterable characters. The list of the common escape sequences are in table 4.5.

Most escape sequences are self-explanatory. The \n is the newline code and causes the printing to move the next line. Since the *fprintf* function does only what is within the formatting string, if the newline was omitted the printing would stop at the last character. The next print statement would then start on the same line. Ending the formatting string with \n forces the cursor to move to the start of the next line - in effect preparing it for the next print statement.

There is a second, similar, escape sequence. The carriage return \r. The carriage return is from the old manual typewriter days. The mechanism upon which the paper rested and where the keys struck the paper was called the

carriage. When a typist neared the end of a line, they would push the carriage back to the beginning to start the next line - thus *carriage return*. Originally the carriage return did not move to a new line but reset to the beginning of the same line. To move to a new line required two commands. Since it is rare in printing to not move to a new line the carriage return escape sequence included the newline. Having two, \n and \r , are redundant but usually still included. The new line escape sequence, \n, is more commonly used.

Another useful escape sequence is the tab \t. Again, a hold over from the old manual typewriters. A typist would set a series of *tab stops* across the length of the page. By pressing the tab key on the typewriter the platen would move so the next key strike would be at the next tab stop. On the computer the tab stops are set a fixed amount across the screen or the paper. Each time the *fprintf* encounters a \t moves to the next tab stop.

The \t works the same way. When the computer encounters a \t in the formatting string it moves the cursor to the next tab stop. While the tap escape sequence is often used to format data so values align on the decimal point, if the width of the numbers vary widely the tab stop issue often disrupts the alignments. Setting the width is usually a better approach.

Warning
The tab escape sequence does not always move the cursor over the same number of spaces. It moves the cursor to the next tab stop. This means it could space over anywhere from one to whatever the tab stop spacing. Because of this when used alone, it may not be a reliable means of aligning data. But can be effective when combined with the width and precision modifiers.

While there are many other escape sequences but there are three that are especially useful. Although they may not be specifically escape sequences - because they do not start with the backslash - the single and double quotes, and the percent sign are still needed when printing but are used for other purposes in the *fprintf* function.

The issue with quotes is that they are already used to start and end a formatting string or a literal. If you use only one quote, MatLab will interpret that as the end of the formatting string. Instead you repeat the quote twice. Since the percent symbol is used to indicate the start of value, the same approach is used to enter the percent sign.

Now that we know how to get the information out of a program, the efficient programming approach is to implement the calculations. But that was covered earlier in the interactive approach. That means we need to address getting the data into the program.

```
function driver()
%   DRIVER driver() is the main or driver function

    %   Demonstrate escape sequences for quotes and percentage symbol
    fprintf('\n');
    fprintf('Don''t use a single quote or a single %% sign\n');
    fprintf('\n');

end
```

```
>> driver()

    Don't use a single quote or a single % sign

>>
```

Figure 4.23: Printing quotes and the percent symbol

4.4 Entering data into the program

Turning data into information requires data, that is any inputs. This can be done in two ways; by hardcoding the values directly into the program, and by having the user enter the data at runtime

4.4.1 Hard coding data

When developing the program it is a time saver to hard code the data directly into the variables that will be used.

Not only will this eliminate the need to constantly enter the data into the program for each variable in the calculations, It will also create a consistent set of inputs. Since you can hard code the inputs from the hand calculation, the results that should print are already known. If what prints is different, then there must be a logic error. Catching these errors at this stage is far easier than searching for them at the end.

4.4.2 Input Function

Hard coding data is useful, especially for constants that are used throughout the program. But they would require the user to edit the script each time the data changes. Instead a method for having the user enter data at runtime is necessary. This is done using the **input** function.

```
function driver()
%   DRIVER driver() is the main or driver function

    %   Hard Coded Data
    dist = 420;
    fuel = 10.0;
    eff = dist ./ fuel;

    fprintf('\n');
    fprintf('%15s%20.1f miles\n', 'Distance: ', dist);
    fprintf('%15s%20.4f gallons\n', 'Fuel: ', fuel);
    fprintf('%15s%20.1f mpg\n', 'Efficiency: ', eff);
    fprintf('\n');

end
```

```
>> driver()

      Distance:         420.0 miles
          Fuel:          10.0000 gallons
    Efficiency:          42.0 mpg

>>
```

Figure 4.24: Hard Coding Inputs

There are two forms for the input function depending upon whether the user will be entering a numerical values - either a floating point value or an integer - or entering a string of text. When entering a numerical value the input function is called as shown in figure 4.25. The variable prompt is usually a string of text, but it does not have to be hard coded into the **input** function. It could be text stored in a variable.

Hardcoding the input data in a model is useful when developing the model. This saves the time of reentering the same values each time the program is run. Further, since the hand solution would have used these same inputs, it is easier to determine if the developing program has any logic errors. This was shown in figure 4.24.

```
variable = input(prompt);
```

Figure 4.25: Syntax of the **input** function for entering a floating point value

But once the program is working and the solution - at least for the single set of inputs - is known to be correct, these hard coded variables should be replaced with calls to the input function as in figure 4.26

```
function driver()
%   DRIVER driver() is the main or driver function

   %   Enter the data for the model
   dist = input('Enter distance driven (miles): ');
   fuel = input('Enter fuel purchased (gallons): ');

   %   Calculate fuel efficiency
   eff = dist ./ fuel;

   %   Print the results
   fprintf('\n');
   fprintf('%15s%20.1f miles\n', 'Distance: ', dist);
   fprintf('%15s%20.4f gallons\n', 'Fuel: ', fuel);
   fprintf('%15s%20.1f mpg\n', 'Efficiency: ', eff);
   fprintf('\n');

end
```

```
>> driver()

Enter distance driven (miles): 380
Enter fuel purchased (gallons): 12.3084

      Distance:         380.0 miles
          Fuel:          12.3084 gallons
    Efficiency:          30.9 mpg

>>
```

Figure 4.26: Entering Numerical Data Using the Input Function

Using the input function as show in figure 4.25 assumes the user is entering a numerical value - a float or an integer. In fact, whatever character is entered is converted from the characters to a numerical value. But if the user mistakenly enters a character or a string of text the interpreter throw an exception and print an error message. To correct this the input function call must indicate that the data to be entered is not numerical but is instead text. This is done by adding the modifier **'s'** to the input function.

Figure 4.27: Syntax of the **input** function for entering a string of text

```
variable = input(prompt, 's');
```

Numerical inputs are limited by white space. This means that if the user attempts to enter several numbers separated with commas or spaces between them the interpreter will give an error. But strings of text allow non-numerical characters such

as symbols or spaces.

```
function driver()
%   DRIVER driver() is the main or driver function

   %   Enter data as a string of text
   name = input('Enter your name: ', 's');

   %   Print the results
   fprintf('\n');
   fprintf('Welcome, %s!\n', name);
   fprintf('\n');

end
```

```
>> driver()

Enter your name: Joe Bfstk!

Welcome, Joe Bfstk

>>
```

Figure 4.28: Entering A String of Text Using the Input Function

4.5 Summary

At its lowest level, a program in an interpreted programming language can be run interactively. The means entering the executable commands one at a time at the command line. While effective this approach is time consuming but also lacks repeatability. Writing a script will alleviate this. A script in an interpreted language is a text file with each command written in the order in which it is to be executed.

In its most simple form, each program consists of three sequential parts; creating the input data, computing using the data, and printing or returning the results or the information. An effective approach to writing the script is to create variables with fixed input and output values. From there you create the outputs.

Printing the results uses **fprintf** with the appropriate formatting string. Once the output is working, the next step is create the computational part of the program with the fixed values in the input variables. Finally, the inputs are created. These are often done using the **input** function.

4.6 Self Test

1. What is the order of operations in arithmetic?
2. What is the result of the following line of code?

$$y = 4 .* 4 ./ 2 + 1$$

3. What is the result of the following line of code?

$$y = 4 .* 4 ./ (2 + 1)$$

4. What is the result of the following line of code?

$$y = 2 .\wedge 3 .* 2 + 16 ./ 4 .* 2$$

5. What is the result of the following line of code?

$$y = 2 .\wedge 3 .* (2 + 16) ./ 4 .* 2$$

6. What does the command **isvarname** do?
7. What does the command **iskeyword** do?
8. What is *echoing*?
9. How is *echoing* implemented?
10. What does the **disp** function do?
11. How are integers and floating point values different?
12. What is precision?
13. What is a formatting string?
14. What is the formatting string for printing an integer? A floating point value? A string of text?

4.7 Projects

1. Write a script that uses **fprintf** with the appropriate formatting to print *Hello World!* on the screen.
2. Write a script in which the user enters two values that are stored in the variables **x** and **y**. It then calculates the mean

$$m = \frac{x + y}{2}$$

   It prints two lines of output. For example, if the user enters 10 and 20, the program prints

X	Y	Mean
10.00	20.00	15.00

3. Write a script in which the user enters two values that are stored in the variables **x** and **y**. It then calculates the

root-mean-square

$$\text{rms} = \sqrt{\frac{x^2 + y^2}{2}}$$

It prints two lines of output. For example, if the user enters 10 and 20, the program prints the result formatted to two decimal places as

X	Y	RMS
10.00	20.00	15.81

4. Write a script in which the user enters the temperature in degrees Fahrenheit and it converts it to degrees Celsius. The transformation formula is

$$^\circ C = \frac{5}{9}(^\circ F - 32)$$

The temperatures should be formatted to show two places of precision. A sample output might be

47.00 degrees F = 8.33 degrees C

5. Write a script in which the user enters the temperature in degrees Celsius and it converts it to degrees Fahrenheit. The transformation formula is

$$^\circ F = \frac{9}{5}^\circ C + 32$$

The temperatures should be formatted to show two places of precision. A sample output might be

27.00 degrees C = 80.60 degrees F

6. A force **F** acting on a body can be decomposed into its x and y components. This requires the use of the sine and cosine functions;

$$F_x = F\cos(\theta)$$

$$F_y = F\sin(\theta)$$

In MatLab, the functions **sind(angle)** and **cosd(angle)** will return the value of the sine and cosine where **angle** is a variable storing the the angle in degrees the force makes at the point.
Write a script in which the user enters two variables; the force **F** and an angle **angle** in degrees. It then calculates the decomposition of the force in the x and y directions. The output should be a table formatted to two decimal places such as

F	Angle	Fx	Fy
100.00	30.00	86.60	50.00

7. A projectile is launched from the ground at an angle θ with an initial velocity of v_0 m/s. The time until it hits the ground is

$$t_f = \frac{2(v_0)_y}{g}$$

where $(v_0)_y = v_0 \sin(\theta)$ and $g = 9.81$ m/sec^2 is the acceleration due to gravity.
Write a script in which the user enters the initial velocity and the initial angle, and the program calculates the time until impact. The output should be a table formatted to two decimal places such as

Velocity (m/sec)	Angle (Degrees)	Impact (sec)
50.00	60.00	8.83

Do not use 9.81 directly in the calculation, but create a variable **g** and set it to the constant 9.81.

8. Three forces are acting at a point at a point. The first is a force of F_1 acting at an angle of $A°$. The second is a force of F_2 acting at an angle $B°$. The third is F_3 acting at an angle $C°$
Write a MatLab program that that analyzes the individual forces and calculates the resultant force. In particular decompose each force into its x and y direction components. Then calculate the resulting x and y components of the resultant force by summing the
component forces. Finally, calculate the overall resultant force and the angle that it would make as a single resultant force.
Have the user enter the three forces, F_1, F_2, and F_3, and the angles that each of these make on the beam, $A°$, $B°$, and $C°$. These are all scalar values.
The formulae for decomposing the angles are

$$F_{1x} = F_1 \cos(A) \qquad F_{1y} = F_1 \sin(A)$$

$$F_{2x} = F_2 \cos(B) \qquad F_{2y} = F_2 \sin(B)$$

$$F_{3x} = F_3 \cos(C) \qquad F_{3y} = F_3 \sin(C)$$

$$F_{Rx} = F_{1x} + F_{2x} + F_{3x} \qquad F_{Ry} = F_{1y} + F_{2y} + F_{3y}$$

$$F_R = \sqrt{F_{Rx}^2 + F_{Ry}^2} \qquad D = \tan^{-1}\left(\frac{F_{Ry}}{F_{Rx}}\right)$$

In Matlab, the command to calculate the trig functions for an angle **A** is **sind(A)** and **cosd(A)**. The inverse tangent is **D = atand(y./x);**
Your program should print the three forces and their angles with the two component forces for each. You will also print the the two components of the resultant force, the overall resultant force and its angle, D.

Procedural Programming 5

A place for everything and everything in its place

Reverend Charles Augustus Goodrich

Many of us approach programming in the same way we approached high school composition - despite everything our teachers told us.

We were told when writing a composition you begin by brainstorming your ideas. From there you create a basic outline of the main topics you would like to cover. You make this outline more detailed by adding subtopics and sub-subtopics. When you have the paper thoroughly outlined - and only then - you write the first draft - by hand. You edit and reedit the drafts until you are comfortable with the current draft. At that point - and not before - you start up the computer and begin typing the final copy.

What do we as programmers - as well as every first semester composition student - actually do? You have a paper to write so you open a new file on the computer and you start typing. You proofread it - mainly for spelling and typos - on the computer. You print it and turn it in.

How does this relate to programing? When we have a task in which we think a computer program would be an effective problem-solving method we tend to start up the computer and start writing code. We do not plan the program. We do not brainstorm or create flow charts. We simply start at the first line and work until the last line.

We can be much more effective and efficient in our coding if we plan out the program. But how do we plan a program?

5.1 Top-Down Design

Big Fleas have little fleas upon their backs that bite 'em. And little fleas have lesser fleas and so ad-infinitum.

Any task can be overwhelming when we only think of getting to the final result. But the reality is that all tasks actually consist of several smaller tasks each of which are more simple when addressed individually. An example is graduating from college.

As an example, every engineering student shares a common goal. To receive a degree in engineering. But how do we accomplish this?
Instead of thinking only of the final goal, break the task down into several smaller tasks. In this case graduating with degree in engineering can accomplished by successfully completing four tasks; successfully complete year one; successfully complete year two; successfully complete year three; and successfully complete year four.
The four subtasks do not explain how to perform the task just what has to be done. Instead we will select one and look at it in more depth - Successfully complete year two. And while you are in your second year there is no reason to be working on tasks that involve year three or year four.

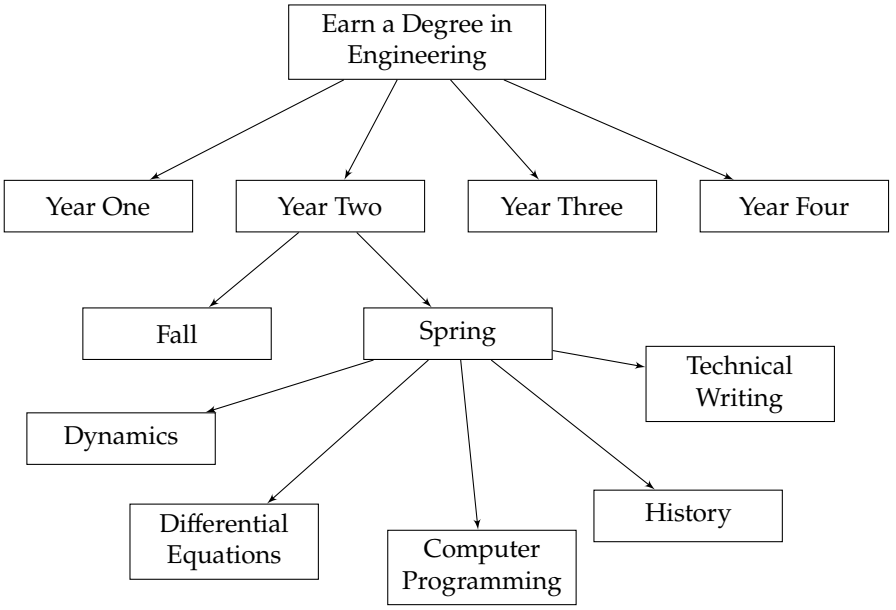


Figure 5.1: Hierarchical Chart for Earning an Engineering Degree.

Further, year two has a Fall and a Spring semester. If it is Spring you have five courses. At any single moment of the semester there is only one class for a particular course that we should be attending.

This is basis of *top-down design;* each task is broken down into the set of subtask that when combined complete the larger task. The subtasks are then broken down into the set of tasks that when completed will have completed the individual subtask. This is repeated until each item is trivial.

The diagram for earning an engineering degree in figure 5.1 is known as an *hierarchical chart*. It is a means of breaking down a task into all of its subtasks. It shows that while there may be a multitude of tasks that will have to be completed, at any single moment there is only one that is active at any given moment.

Top-down design provides a means to compartmentalize these processes so that we can plan what tasks will need to be done.

5.1.1 Hierarchical Chart

A means of planning the top-down design involves creating an *hierarchical chart*. The hierarchical chart is similar to a flow chart. But unlike a flow chart where the items show the sequence of steps that will be followed, the hierarchical chart lays out each task and its subtasks. It does not provide the order of running the tasks but instead it provides a means of planning and identifying the tasks that will have to be done.

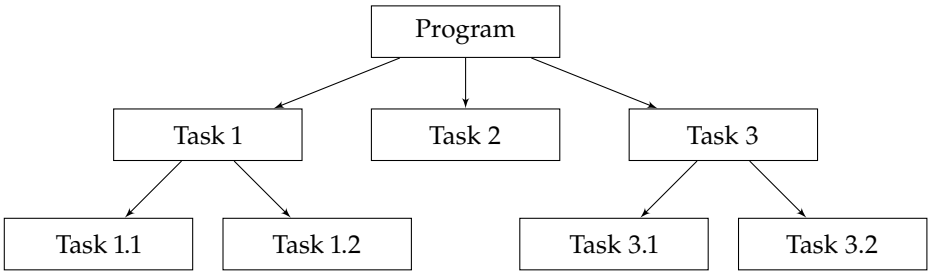


Figure 5.2: Hierarchical Chart for Top-Down Programming.

An hierarchical chart is beneficial in any project management - it is actually similar to a PERT (*Project Evaluation and Review Technique*) diagram. Our interest is using it to implement top-down design to simplify programming an algorithm, and in the end making the process more efficient.

Hierarchical Chart
An hierarchical chart is a diagram in which each task is broken down into the smaller subtasks that comprise it.

In creating the hierarchical chart the program - or the project - is the lead task. From there and engineer would divide the program into the set of unique, or non-overlapping, tasks that will complete the project. Each of these would then be divided again into the next set of unique subtasks when completed will satisfy the task directly above it. Once every subtask is defined, the hierarchical charts can be used to

create a flow chart by determining the order and dependencies for each task.

While the hierarchical chart is a useful tool for any project management, it is just as useful in creating an algorithm and thus a program. In this application each of the tasks will be *functions*.

5.1.2 Functions

While different programming languages call them by different names - such as methods, subroutines, or procedures - functions are the implementation of top-down design. Regardless of the name their purpose is the same. A function performs some specialized task in a program.

Function
A set of code that performs a particular task.

The purpose of functional programming, or what is often called *procedural programming*, is that it enables us to use top-down design to simplify programming.

In a small way, we have already been writing functions in our programs. The program itself is a function. It performs all of the tasks that we want the program to perform. There is not much reason to think of a program as a single function, but to implement top-down design we will now need to so. Instead of the entire program in one function, we will now consider the lead, or main, function as a driver of the rest of the program. Our goal will to eliminate all of the details from this function and instead let it act as a project manager; calling each function when it is needed and doing only the most basic computing itself.

Named Function
An important aspect of the driver function in MatLab is that the name of the function is the same as the name of the script file. Thus we may also call it a *named function*.

This analogy of driver as project manager lends itself to a flow chart, figure 5.3, of the program from the perspective of the driver. All that the driver does is call other functions.

Taking the executable details of the program away from the driver will provide additional benefits. A program will now consist of multiple functions that

- can be divided between many members of a development team. Each member might be responsible for only one function but when all joined into the program will provide the needed functionality.
- can be used multiple times. If a function only performs a single task, but that task is needed multiple times it only needs to be coded once.

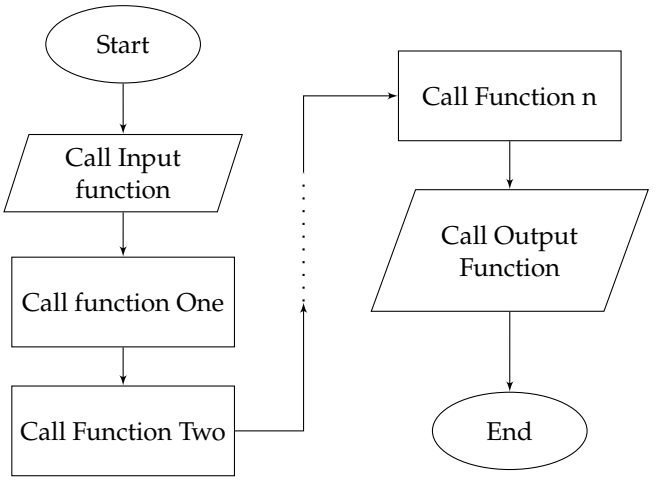


Figure 5.3: Flow Chart for the Driver as Project Manager

- can be easily revised. It often occurs that a function will be have to be updated. When this occurs only a single portion fo the program will have to be changed.
- can be more easily debugged. Since each function performs only a single task it will by its nature be a much smaller part of the program. If there is an error in it then the error is likely to be more easily found and corrected.

The functions that we will be using can be separated into two types; built-in functions and user-defined functions.

Function Calls and Definitions
It is important to differentiate between the function call and the function definition.

- Function Call
  To use a function you need to call it. The function call is a single line with the function handle, a set of of input variables, and a set of output variables.

- Function Definition
  The function definition is the list of instructions or executable statements that are run when the function is called. For a local function it will start with the keyword function and finish with the keyword end.

5.2 Built-In Functions

Luckily for us there are a multitude of functions that are built into MatLab. We could not print without **disp(...)** or **fprintf(...)**. We could not enter data into the program without **input(...)**. And without **sqrt(...)** we would have to write our own code to calculate the square root of a number. These are examples of built-in functions.

5.2.1 Function Calls

In MatLab, implementing a built-in function is done by making a *function call* as in figure 5.4. The function call has three parts; the function name, the list of output variables, and the list of input parameters.

Function Call
A function call is a program statement that passes control of the program over to a subprogram, or function. The function then performs a task and returns control to the point in the program from which the function was called.

The output variable list and the input parameter list are optional. If the function does not return values to its driver function then there is no need for the output variables. Similarly, if the function does not require the driver to pass data to it for it to operate then there is no need for the input parameters. It is important that the parentheses be included even if the function does not require input data. Built-in

```
% Syntax of a MatLab Function Call
[output vars] = function_name(input vars);
```

Figure 5.4: Syntax of a Function Call

functions can be called directly from the command line in the same way that calculations can be done from the command line. The **date( )** function does not require inputs or outputs so it can be called directly. Computational functions will

```
>>  date( )

ans = 24-May-2019
```

Figure 5.5: Calling the date( ) function from the command line

more often have both inputs and outputs. As with simple

```
>>  x = sqrt(42.0)
x =
    6.4807
>>
```

Figure 5.6: Calling *output variable* = sqrt(*input parameter*) from the command line

calculations, function calls from the command line can be convenient but their value is in how they are run from a script.

```
function driver( )
%   DRIVER driver( ) is the main or driver function for the program

    % Enter data
    x = input('Enter the value for the square root: ');
    s = sqrt(x);
    fprintf('sqrt(%0.3f) = %0.3f\n', x, s);

end
```

```
>> driver()

Enter the value for the square root: 42
sqrt(42.000) = 6.481
```

Figure 5.7: Calling *output var* = sqrt(*input par*) from a script

5.2.2 Built-In Functions

MatLab has built-in functions that address many calculations. The basic trigonometry functions are listed in table 5.1. There

Table 5.1: Trigonometric Functions

Function	Example	Description
sin	y = sin(radians)	Sine of argument in radians
cos	y = cos(radians)	Cosine of argument in radians
tan	y = tan(radians)	Tangent of argument in radians
sind	y = sind(degrees)	Sine of argument in degrees
cosd	y = cosd(degrees)	Cosine of argument in degrees
tand	y = tand(degrees)	Tangent of argument in degrees
asin	radians = asin(y)	Inverse sine in radians
acos	radians = acos(y)	Inverse cosine in radians
atan	radians = atan(y)	Inverse tangent in radians
asind	degrees = asind(y)	Inverse sine in degrees
asind	degrees = acosd(y)	Inverse cosine in degrees
atand	degrees = atand(y)	Inverse tangent in degrees
hypot	c = hypot(a, b)	Square root of the sum of squares (Pythagorean Theorem)

is also a set of general algebraic functions in table 5.2. With the exception of the **hypot** function the input for each of these is a single value and each return a single value as well.

Table 5.2: Algebraic Functions

Function	Example	Description
exp	y = exp(x)	Base e exponential, $y = e^x$
log	y = log(x)	Natural logarithm
log10	y = log10(x)	Common logarithm (Base 10)
log2	y = log2(x)	Base 2 logarithm
pow2	y = pow2(x)	Base 2 power
sqrt	y = sqrt(x)	Square root
abs	y = abs(x)	Absolute value

Overloaded Function
An overloaded function is a function that while having a single function name performs different tasks depending upon the function signature.

But functions are often multivariate. The **hypot** function is an example. When called the program passes two values to the function representing the shorter two legs of a right triangle. The function calculates the length of the third side - the hypotenuse - and returns that value to the driver.

It is possible for a function to return more than one value. The conversion functions in table 5.3 have several functions that return two and even three values.

There are two functions in table 5.3 that appear twice; **cart2pol** and **pol2cart**. These are overloaded functions; functions that have the same function name but depending upon the its function signature it does a different calculation or performs a different task. Each function has a unique

```
function driver()
%   DRIVER driver( ) is the main or driver function for the program

    % Enter data
    x = input('Enter the first leg: ');
    y = input('Enter the second leg: ');
    z = hypot(a, b);
    fprintf('The right triangle has sides %f, %f, and %f\n', x, y, z);

end
```

```
>> driver()

Enter the first leg: 4
Enter the second leg: 3
The right triangle has sides 4.000, 3.000, and 5.000
>>
```

Figure 5.8: Passing two values to a multivariate function

Table 5.3: Conversion Functions

Function	Example	Description
deg2rad	**d = deg2rad(r)**	Convert angle from degrees to radians
rad2deg	**r = rad2deg(d)**	Convert angle from radians to degrees
cart2pol	**[theta rho] = cart2pol(x, y)**	Transform Cartesian coordinates to polar or cylindrical - 2D
cart2pol	**[theta rho z] = cart2pol(x, y, z)**	Transform Cartesian coordinates to polar or cylindrical - 3D
cart2sph	**[azimuth elevation r] = cart2sph(x, y, z)**	Transform Cartesian coordinates to spherical
pol2cart	**[x y] = pol2cart(theta rho)**	Transform polar or cylindrical coordinates to Cartesian - 2D
pol2cart	**[x y z] = pol2cart(theta rho, z)**	Transform polar or cylindrical coordinates to Cartesian - 3D
sph2cart	**[x, y, z] = sph2cart(azimuth, elevation, r)**	Transform spherical coordinates to Cartesian

Function Signature The function signature is a unique representation of the function call that indicates which function should be executed.

signature consisting of the function name and the number of input parameters.

The number of output variables listed is the maximum number that can have values be returned from the function. It is possible to return fewer - or even zero - but the result may not be as expected. The function itself determines the number and type of values to return. It could be that a single variable in the output list could return a single value, or it could return a vector with multiple values, or could just result in a run time error.

Because of this, unless you know how the function will respond it is best to have the same number of variables as the function is designed to return.

While returning fewer values is useful, there is an issue. You cannot pick and choose which value to return. If you want the

```
function driver()
%   DRIVER driver() is the main or driver function for the program

    % Enter data
    x = input('Enter the x coordinate: ');
    y = input('Enter the y coordinate: ');

    [theta, radius] = cart2pol(x, y);

    fprintf('\t\t(x, y)\t\tradius\t\tangle (rad)\n');
    fprintf('\t(%0.2f, %0.2f) %10.2f %10.2f\n)', x, y, theta, radius);

end
```

```
>> driver()

Enter the x coordinate: 4
Enter the y coordinate: 3
        (x, y)          radius        angle(rad)
     (4.00, 3.00)        5.00             0.64
```

Figure 5.9: Returning multiple output values

first only then you only need a single variable on the left of the assignment operator. But if you only want the second you must have two variables within the square brackets on the left. If you have only one - and even if you name with the second variable - it will receive the value in the first return variable.

Warning
If you use fewer values than what is shown in the return variable list you may get unexpected results. Unless you know how it the function handles this it is best to provide variables for all of the return values.

5.2.3 Finding an appropriate built-in function

With the thousands of built-in functions that are available how do you 1. know what functions are available, and 2. know how to use the function?

MatLab provides you with many functions that are meant to be called from the command line, as compared to being coded into a script. Because of the way that they are commonly used these functions are usually given the moniker *commands* instead of *functions*. Two of these system command can help with finding and using built-in functions. They are **lookfor** and **help**.

lookfor
lookfor is a system command that searches the function help files for a particular keyword.

Searching for a function

With the thousands of functions that are available how might you find the one that you need? **lookfor** is a system

command for searching for a functions.

```
% Syntax of the lookfor command
>> lookfor keyword
```

Figure 5.10: Syntax of the **lookfor** command

Each built-in function contains a block of comments that are intended to provide help to the programming. The **lookfor** command functions by searching these help files for the keyword that you entered. If it finds the keyword then it returns the function name in which it was found and also a short description of the function (figure 5.11).

```
>> lookfor cosine
acosd   Compute the inverse cosine in degrees for each element of X.
cosd    Compute the cosine for each element of X in degrees.
acos    Compute the inverse cosine in radians for each element of X.
acosh   Compute the inverse hyperbolic cosine for each element of X.
cos Compute the cosine for each element of X in radians.
cosh    Compute the hyperbolic cosine for each element of X.
>>
```

Figure 5.11: Example of the **lookfor** command

Help with a function

The **lookfor** command will provide you with a list of possible functions but it does not tell you how to implement the function in your script. What input parameters does it require? What are the outputs values? These can be found using the **help** command.

```
% Syntax of the lookfor command
>> help function name
```

Figure 5.12: Syntax of **help** command

As an example, figure 5.13 shows the use of the **help** command for the exponential function, e^x.

The built-in functions are a feature of the top-down design in MatLab but there purpose is not so much to break tasks down

```
>> help exp
exp Exponential.
    exp(X) is the exponential of the elements of X, e to the X.
    For complex Z = X+i*Y, exp(Z) = exp(X) * (COS(Y)+i*SIN(Y)).

    See also expm1, log, log10, expm, expint.

    Reference page exp
    Other functions name exp
```

Figure 5.13: Example of the **help** command

5.3 Local Functions

"If you want it done right, do it yourself."
Napolean Bonaparte

Despite there being hundreds of built-in functions, and probably hundreds of thousands more that can be found in repositories, there is a need to be able to write your own.

Local Function
A local function is a user defined function that is written outside of any other function. It can be called by any other function in the same script.

Top-down design demands that the program be broken down into a finite number of trivial tasks. These are not just tasks for which someone else has already written a function.These include specialized printing functions. Functions to perform inputs and error check those inputs. And most often, functions that simply perform some small set of calculations. It often makes sense to hide away the details of these functions and instead just use a function call.

Warning A common error in writing local functions is to embed them within another function. It is necessary that the entire local function, from the keyword **function** to the keyword **end** be completely outside of any other function. This includes the driver.

The most common type of the do-it-yourself function is a *local function*. This is a function whose function definition takes a form similar to that of the driver function. But the local function definition is written outside of the driver function or any other local function. Because of the importance of their being separate from all other functions, a common approach is to write each one after the current final keyword **end** of every other function.

A local function will have two parts; a function call and a function definition. The function definition is the set of executable statements for the function. It will take the same format as the driver function but must be completely outside of any other function.

Output Variables

The local function is very similar to the driver function that we have been using from the start. It begins with the keword

function and concludes with the keyword **end**. On the first line, after the keyword **function** on the left side of the assignment operator is the list of output variables. There does not have to be an output variable and there is no upper limit to the number used. If there is not a return variable then it, and the **=** are omitted. This is the usual case in the driver function. If there is more than one then they listed separated by either commas or a space within a pair of square brackets.

```
function [output vars] = function_handle(input vars)
%   Help comments for the local function

    %   Operations for the local function

end
```

Figure 5.14: Syntax of a local function

Function Handle
The function handle creates an association to the function. While it looks like variable name, the function handle is a structure that includes information about the function. This includes the function name, the type of function, and the file in which the function is written.

structure = functions(@f) The built-in function **functions** returns the details of the function handle that is passed to the function. This includes function name, the type of function - whether built-in, local (scoped), nested, or anonymous, the file in which the function is stored, and the parent function for the function.

Function Handle

Following the output variable list is the function name or what is known as the *function handle*. The function handle is not a variable or simply a name but is in fact a structure. Each item in the structure contains information about the function including its name, type of function, and the file, if any, in which the function is written. This is information is available by calling a function called **functions**. You can do this from the command line as in figure 5.15.

The function handle makes is possible to pass a function to another function as in input (section 5.6). The naming convention for a function handle is the same as it is for variables.

```
>> functions(function_handle)

     function: 'function_handle'
         type: 'scopedfunction'
         file: 'functionsExample.m'
    parentage: {'driver'}
```

Figure 5.15: Example of the functions command

Input Variables

The third, and final part of the first line of a local function is the input variable list.

The input parameters are a comma delimited list of variables that will be receiving data from the driver. They are not, however, the variables from the driver. Instead they are new variables that are created when the function is called. They receive a copy of the value in the variables in the driver.

Input Variables The input variables do not currently exist but will be created when the local function is called. The values in these variables are copies of the values in the variables in the function call.

There are no limits on the number of input variables. You may create local functions that have no input variables, or hundreds of variables, or any number in between. Figure 5.16 shows a local function with two input parameters.

```
function driver( )
%   DRIVER driver( ) is the main or driver function

    % Input data
    x = 5;
    y = 3;

    % Call the local function
    z = local_function(y, x);

    % Print the results
    fprintf('%0.3f .^ %0.3f = %0.3f\n', x, y, z);

end
%   All local functions are written outside the driver
function r = local_function(base, power)
%   LOCAL_FUNCTION r = local_function(s) is an example of a local
%   function.Two values are passed in and one is returned

    %   Calculate the value of r
    r = base.^(power);

end
```

```
>> driver( )
3.000 .^ 5.000 = 243.000
>>
```

Figure 5.16: Example of a local function

Comment It is not necessary that the variable names in the input parameter list be the same as the variables in the function call. It is often recommended that you provide different variable names for these variables to differentiate between the variable in the function call and the variable in the local function.

While most local functions are intended to perform specialized calculations, they are also used for customizing inputs and outputs.

Output functions are local functions that print. It could be printing descriptions as we will see in the **print_header** function, or printing the results of calculations.

Input functions are the opposite. They enable the user to enter data into the program. This could be as simple as

calling **input** or they could be designed to collect data from equipment, sensors, or network servers. Input functions will often include error checking to ensure the data being used in the program meets the proper criteria or constraints.

Output Functions

An output function is a type of local function that prints but does not actually return a value. A common example is a splash screen function (figure 5.17).

```
function driver()
%   DRIVER driver( ) is the main or driver function

    % Print the Splash Screen
    print_header();

end
%   All local function definitions are written after the driver
function  print_header()
%   PRINT_HEADER print_header() prints start up information

    %   Print the splash screen
    fprintf('\n');
    fprintf('This is the Splash Screen \n');
    fprintf('It acknowledges the programmer, and provides \n');
    fprintf('distraction while the program is loading\n');

end
```

```
>> driver()

This is the Splash Screen
It acknowledges the programmer, and provides
distraction while the program is loading
>>
```

Figure 5.17: Example of a splash screen function

This splash screen function does not receive any inputs or return outputs. It just prints. We can make a change to it so that it does receive a set of input parameters and then adjusts what it prints based upon the inputs. This is a form of customization that will make the function portable, that is generalized so that it can be used in many other programs without any changes. The items that do change are passed to the function instead of hard coded into it.

In this example, figure 5.18, the splash screen will print the

programmer's name, the date, and a short description of the program. If we copy this function into a different program, all that we need to do is change the date and description in the function call. The function definition does not change at all.

```
function driver()
%   DRIVER driver( ) is the main  function

    % Set the inputs to the splash screen
    name = 'Joe Bfstk';
    current_date = '28 May 2042'
    desc = 'Showing off the Splash Screen';
    % Print the Splash Screen
    print_header(name, current_date, desc);

end
%   All local functions are written after the driver
function print_header(n, d, description)
%   PRINT_HEADER print_header(n, d, desc) prints start up information

    %   Print the splash screen
    fprintf('\n');
    fprintf('Name: %s\n', n);
    fprintf('Date: %s\n', d);
    fprintf('Desc: %s\n', desc);
    fprintf('\n');

end
```

```
>> driver()

Name: Joe Bfstk
Date: 28 May 2042
Desc: Showing off the Splash Screen

>>
```

Figure 5.18: Splash screen function with input parameters

In this customized version of **print_header** the data is loaded into three variables in the driver. The values are then passed to the function where three new variables are created and the data from the driver is copied in to them.

This approach, writing a function that is passed data to be printed while not returning data to its driver is useful for printing results. By placing all of the output printing together it can be handled by a single function call while at the same time being formatted to make all of the output the most effective for the user.

```
function driver()
%  DRIVER driver() is the main function for the program

    % Enter data
    dist = 400.0;
    vol = 28.0;

    % Calculate the fuel efficiency
    lpk = 100 .* vol ./ dist;

    % Print the results
    print_results(dist, vol, lpk);

end
%   Local functions are written after the driver
function print_results(d, f, e)
%   PRINT_RESULTS print_results(d, f, e) prints the outputs
%   as a formatted table

    %   Print the results as a table
    fprintf('\n');
    fprintf('Distance: %13.1f km\n', d);
    fprintf('Fuel: %17.1f liters\n', f);
    fprintf('Efficiency: %12.2f liters/100 km\n', e);
    fprintf('\n');

end
```

```
>> driver()

Distance:    425.3 km
Fuel:         29.7 liters
Efficiency:    6.98 liters/100 km

>>
```

Figure 5.19: Example of an output function.

Abstraction Abstraction is the process of reducing the complex nature of a program into smaller, and hopefully simpler steps. It is done using top-down design and the use of functions.

With functions that perform calculations and functions that are meant to to print, all that is left is a function that is used for entering data into the program.

Input Functions

A means of entering data into a program has already been provided with the built-in function **input**. This function prompts the user and then receives the user's data entry, storing it in s variable. It differentiates between numerical data and strings of text. It is, however, limited.

The **input** only accepts user entered data - it stops the execution of the program and waits for the user to type on the keyboard. It also requires that each time a value is to be entered it a separate call to **input** must be made. What if there are multiple data values to be entered? Or if the data is being sent from an alternative device such as a sensor or server, or from a network feed?

If the data is to be user entered, **input** does not provide any error checking - an important part of any data entry but especially so for user entered data. .

An improvement to the **input** function is to create a *wrapper* function. This is a function that might combine multiple calls to **input** so several data values can be entered while at the same time moving the details out of the driver. Not limited to data entry, their use is to provide abstraction and thus programming convenience.

Wrapper Function A wrapper function is a function whose purpose is to call one or more other functions while performing a minimum of computation. It is a means of abstraction in that the implementation details are hidden from the outside.

The wrapper function can be used to combine multiple inputs into a single function while at the same time providing error checking all while keep the details out of the driver. Adding in a wrapper function to the fuel mileage program is shown in figure 5.20.

As the example progressed we were able to remove details from the driver and replace them with simple function calls. By the end the driver did little more than just call other functions. All the details are handled in the local functions. This is the *abstraction* that was described earlier.

Local functions are written outside of the driver function. This means that the function definition of a local function may not be within any other function. The are also designed so they may be called from every other part of the program. This includes both the driver function - the function with the same name as the script file - and any other local function.

```
function driver()
%   DRIVER driver() is the main or driver function

    % Enter data
    distance = get_data('Distance driven: ');
    volume = get_data('Fuel purchased: ');

    % Call function to calculate efficiency
    lpk = fuel_efficiency(distance, volume);

    % Print the results
    print_results(distance, volume, lpk);

end
%   All local functions are written after the driver
function x = get_data(prompt)
%   GET_DATA get_data(prompt) is a wrapper for input data

    %   Call the input function
    x = input(prompt);

end
```

```
>> driver()
Distance (km): 400.0
Fuel purchased (liters): 29.7

Distance:      425.3 km
Fuel:           29.7 liters
Efficiency:      6.98 liters/100 km

>>
```

Figure 5.20: Example of a wrapper function

5.4 Nested Functions

Nested Function A function that is defined completely within another function is *nested* within that function. They are exist, and are thus callable, from within their parent function.

There are programming applications in which applying abstraction suggests that you write a function, but the function has only a limited use. Perhaps it will only be used in a single function, or it requires input values that only exist within a function, or it uses parameters that must be defined before the function can even be defined. In these cases the use of a *nested function* can be useful.

A primary difference between a nested function and a local function is how it uses variables. Any variable that is created in the parent of the nested function is available to the nested function. In a local function only those variables created within the function are available to the function.

```
function driver()
%   DRIVER driver( ) is the main or driver function

    % Enter data
    r = input('Enter the radius of the polygon (m): ' );
    % Nested function to calculate area
    function a = areaPolygon(n)
        % Area of a regular polygon
        a = (r.^2) .* n .* sind(360 ./ n) ./ 2.0;
    end

    % Print areas
    fprintf('Triangle: %10.2f m^2\n', areaPolygon(3));
    fprintf('Square: %14.4f m^2\n', areaPolygon(4));
    fprintf('Pentagon: %10.2f m^2\n', areaPolygon(5));

end
```

```
>> driver()
Enter the radius of the polygon (m): 5

Triangle:     32.48 m^2
Square:       50.00 m^2
Pentagon:     59.44 m^2
>>
```

Figure 5.21: Example of a nested function

This can be seen in the example in figure 5.21. The user enters the radius of the polygon in the driver. When they call the area function they pass the number of sides - but not the radius. The radius, being created before the nested function in the parent function, is automatically available to the nested function.

Because of the variable behavior we could write the nested function with no input variables at all. In fact, there are many times when this is done. The reason for the nested function is then a matter of repeatability. If there is a set of code that needs to be implemented multiple times, but only in a single function, then a nested function can greatly shorten the amount of coding that goes into the function.

5.5 Anonymous Functions

There is a third type of function, that while often confused with nested functions, it is more like a local function. But in reality. they are really neither. These are *anonymous functions*.

Figure 5.22: Syntax of an anonymoust function

```
function_handle = @(input parameters) expression;
```

Anonymous Function An anonymous function is a single executable line of code that is defined within another function.

An anonymous function is defined almost as if it were a single line expression. The difference is the addition of a declaration of a set of input parameters in the anonymous function.

As an example, recreate the area of the polygon function but this time using an anonymous function.

```
function driver( )
%   DRIVER driver( ) is the main or driver function

    % Enter data
    r = input('Enter the radius of the polygon (m): ' );
    % Anonymous function to calculate area
    areaPolygon = @(n, r) (r.^2) .* n .* sind(360 ./ n) ./ 2.0;

    % Print areas
    fprintf('Triangle: %10.2f m^2\n', areaPolygon(3));
    fprintf('Square: %14.4f m^2\n', areaPolygon(4));
    fprintf('Pentagon: %10.2f m^2\n', areaPolygon(5));

end
```

```
>> driver( )
Enter the radius of the polygon (m): 5

Triangle:     32.48 m^2
Square:       50.00 m^2
Pentagon:     59.44 m^2
>>
```

Figure 5.23: Example of an anonymous function

In this example, an anonymous function is defined within a parent function. Once done it is callable from that same function.

There is an important difference between the nested function example in figure 5.21 and the anonymous function example in figure 5.23. It is in the treatment of the two variables **r** - radius, and **n** - number of sides. In the anonymous function both the radius and number of sides need to be passed to the function as input paramters. But in the nested function the radius variable is not passed to the function but instead is inherited from the parent function. On the surface this appears to be a minor difference, but it is an important

difference. The need to do this is the primary difference between the anonymous function and a nested function.

5.5.1 Anonymous functions or nested functions

Anonymous functions and nested functions share a couple of similarities. They are both defined within another function. Because of this they are only callable from the parent function or any nested or anonymous functions at the same level. An anonymous function cannot be called from other local functions in the program.

The difference is one that separates local functions from nested functions; *encapsulation*. Recall that a local function does not have access to variables in other functions while a nested function has a limited access to the variables that are declared in the parent function.

Variables in the parent function and variables in the anonymous functions are hidden from each other. In this way anonymous functions behave more like a local function. If you need data in an anonymous function then it must be passed to the function as an input parameter.

Similarly, any information that is created in the anonymous function must be returned to the calling function as the return variable. Since an anonymous function does not have an explicit return variable like local and nested functions, the return variable is the result of the anonymous function's computation.

Anonymous or Nested There are similarities and differences between anonymous functions and nested functions.
Similarities

- Defined within a parent function
- Callable from the parent.

Differences

- Nested functions inherit variables from the parent function.
- Variables in anonymous functions must be passed to the function (encapsulation).
- Anonymous functions can be parameterized.

5.5.2 Parameters in Anonymous Functions

While an anonymous function does not have direct access to variables in its calling function, there is an exception. Any variable that is declared before the anonymous function can be used as a constant or a parameter in the anonymous function. These parameters are fixed for the lifetime of the function.

In the example in figure 5.24 the parameters for the quadratic equation are set before the function was defined. Since they do not appear in the variable list for the anonymous function they are made to be fixed parameters.

Once any parameters in an anonymous function are set they cannot be changed. You can change the variables that were

Warning Parameters in an anonymous function are fixed when the function is defined. While the variables in the calling function can be changed after the anonymous function is defined the parameters in the anonymous function will not.

```
function driver()
%   DRIVER driver() is the main or driver function

    % Enter data
    a = input('Enter the first coefficient:  ' );
    b = input('Enter the second coefficient:  ' );
    c = input('Enter the third coefficient:  ' );
    x = input('Enter the value of x: ' );
    % Anonymous function to calculate quadratic
    quadratic = @(x) a .* (x.^2)  + b .* x + c;

    % Print value of quadratic function
    fprintf('f(%0.2f) = %0.2f n', x, quadratic(x));

end
```

```
>> driver()
Enter the first quadratic coefficient: 3
Enter the second quadratic coefficient: -2
Enter the third quadratic coefficient: 5
Enter the value of x: 3

f(3.00) = 26.00
>>
```

Figure 5.24: Example of parameters in an anonymous function

used to create the anonymous function after the function was created but doing so will not change the function.

5.5.3 User entered anonymous functions

It often occurs in modeling that a mathematical function is not known when the program is being written. Or the function may change each time that the user runs the program. A solution is a user entered function.

A powerful use of anonymous functions is that instead of requiring they be hard coded - like a local function - into the program, they can be entered by the user at runtime.

f = str2func(s)
When entering a function as a string of text it is necessary to create a function handle for this function. This is performed with the built-in function **strcat**. The input to this function is a string of text, **s**. The output is a function handle, **f**.

Since strings of text can be entered at runtime, if the text can be transformed from a set of characters that form a mathematical expression into an anonymous function it can be called just like a function that was hard coded into the program. This is done with the **str2func** function.

Once the string expression has been transformed into a function it can be used like any other anonymous function.

That is it can be called or passed to other functions.

```
function driver( )
%   DRIVER driver( ) is the main or driver function

    % Get the anonymous function
    f = get_function('Enter the function expression: ');
    % Enter the input value
    x = input('Enter an input value: ');

    % Evaluate the anonymous function
    y = f(x);

    % Print results
    fprintf('f(%0.2f) = %0.2f n', x, y);

end
function f = get_function(prompt)
%   GET_FUNCTION get_function(prompt) allows
%   the user to enter an expression transformed
%   into an anonymous function

    %   Enter the string
    s = input(prompt, 's');
    %   Concatenate an @(x) on the front
    s = strcat('@(x)', s);
    %   Transform s into a function handle
    f = str2func(s);

end
```

```
>> driver( )
Enter the function expression: x.^2 - sin(x ./ 2)
Enter the input value: 2

f(2.00) = 3.16
>>
```

Figure 5.25: Example of a user entered anonymous function

It is possible to write a function whose purpose is to have the user enter an expression that will become an anonymous function. In this approach the user enters a string of text. The text is then transformed into a function which is callable as any other anonymous function.

The function, **f = get_function(prompt)**, prompts the user for a string of text. It could have the user enter the input parameter list such as **@(x)**, but in this case it is added to the front of the expression string.

String Concatenation
String concatenation is the process of joining two strings together.

After the input parameters are concatenated onto the

expression, the final step is for the string to be transformed into a function handle. This is performed with the built-in function **str2func(string)**.

The ability to have the user enter the function at run time can be quite useful. Imagine that you have a program in which you estimate the first derivative of a function at at particular point. The problem is that you do not what the function will be until you run the program. It might be a quadratic, or a trig function, or something completely different. It would be useful to have the user enter the function after the program starts running and it then estimates the derivative of the function that was entered.

5.6 Passing a function handle to another function

The previous idea of estimating the derivative of a function can be abstracted by creating a local function that does the estimation. If the anonymous function could be passed to the derivative function.

An example of passing a function to a function involves estimating the derivative. The person writing the code would not know what function needs to differentiated - if they did they would just solve it then.

Instead they can generalize the program so that the function can be created at run time, and then passed to the derivative function.

In the example (figure 5.26) the user enters the function expression as text. The function handle along with a tangent point and a step size is passed to the **derivative** function. The derivative function uses the common formula for estimating the derivative of a continuous function at a point by calculating the slope of the secant line.

```
function driver( )
%   DRIVER driver( ) is the main or driver function

    % Get the anonymous function - use the function definition from
    before
    f = get_function( );
    %   Enter the input value
    x = input('Enter the tangent point: ');
    % Estimate the derivative
    fPrime = derivative(f, x);

    % Print results
    fprintf('f''(%0.2f) = %0.2f n', x, fPrime);

end
function m = derivative(f, x)
%   DERIVATIVE m = derivative(f, x) is a local function that
%   estimates the derivative. This demonstrates passing a function
%   handle to another function

    dx = 0.01;  %   Small step size
    m = (f(x + dx) - f(x)) ./ dx;   %   Estimate of derivative

end
```

```
>> driver( )
Enter the function expression: x.^2
Enter the tangent point: 1

f'(1.00) = 2.01
>>
```

Figure 5.26: Estimating the derivative of a user entered function

5.7 Scope and Lifetime

When a function, whether it be a local function or a nested function, or an anonymous function, a block of memory is reserved for the function. This includes space for any variables that are created in the function. When the function exits, returning any values to the calling function the memory is erased and with it the function and its variables. The availability of the function to the program is a matter of its *scope* and its *lifetime*.

5.7.1 Scope

In terms of functions and variables the scope is much like a person's scope. For each of us, our scope is where we can be accessed; where can someone speak with us or call us or send us a message. We can control our scope by setting where we will be allowed to be contacted. Perhaps you will allow someone to meet with you or call you in your office, but not at home. Your scope is then your office. Another term for the scope is the *visibility*.

Scope
The scope of a function or variable is the part of the program that can call or have access to the function or variable.

Scope of functions

With respect to a function the scope of the function is from where in the program the function be called. This is often refined to which other functions may call it. There is a clear difference between the scope of a local function and the scope of a nested function.

Recall that a local function can be called from the program driver or from any other local function. Thus the scope is any other local function or the main program driver function.

But nested functions can only be called from their parent function or other nested functions that are at the same level in the parent function. The scope for a nested function is thus its parent function and other nested functions in the parent function at the same level.

Scope of variables

So how is this different for variables?

Variables created within a local function or within the driver function are, by default, only accessible within the function. Thus their scope is the function in which they are declared.

Variables in nested functions are treated a bit differently. Recall that variables created in a function are available to any nested function that was defined within the function. Because of this a variable that is created in a function, whether it be a driver function, local function, or nested function, adds to its scope any nested functions that are defined within it.

5.7.2 Lifetime

For a person, scope is where the person can be contacted or is visible to those who want to contact them. Their lifetime is when they become visible to the time that they are no longer visible; or birth to death.

Lifetime
The lifetime of a function or a variable is when it is visible. It is normally the time from when the function is defined or the variable is declared until either the end of the function of the end of the program.

This is similar for functions and variables. Whereas scope is the visibility, that is where it is available in the program, lifetime is when. It is a chronological measure. Scope is where the function or variable is visible, lifetime is when it is visible

Lifetime of a function

For a local function the lifetime of the function is the from when the program begins to when it ends. For a nested function the lifetime is from when the parent function is called until it returns.

Lifetime of a variable

The lifetime of a variable is much like the lifetime of a nested functions. The default behavior of variables are that they are created when they are first declared - that is memory is reserved and normally a value is stored in memory for the variable, and are deleted when the function in which they are declared returns. Thus their lifetime is from when the variable is created until the function in which it is created returns control to the calling function.

If the variable is declared in a local function and the function in turn calls another local function, the lifetime of the variable continues but not the scope. This is a result of the function no longer being active, having passed control to the other local function. But if the function calls a nested function then the variable continues with both its lifetime and its scope since variables are accessible within the parent's nested function.

5.8 Encapsulation

Local functions and anonymous functions operate on the concept of *ecapsulation*. This means that all variables are by default hidden within the function in which they are created. Further, they are inaccessible from outside of the function unless they are explicitly passed to another function.

Encapsulation
While normally a term applied to object oriented programming, encapsulation is a mechanism for restricting access to variables that are created in a function. It requires that all outside data that is needed in a function be passed to the function through the input parameters, and all information be returned to the driver function through a return variable. It is also known as *information hiding*.

Encapsulation keeps all of the variables that were created outside of a function outside. It also keeps all variables that were created inside of the function inside. This is meant to provide stability. So how does this operate?

Pass By Value
A parameter is passed by value when the calling function has a variable that contains data. At the same time the function being called creates a new, separate and independent, variable. The value in the calling function variable is then copied into the new function variable. If the function modifies its local variable, the change has no effect on the calling function variable.

> *Imagine yourself as a function. You sit in a locked room with only a single window. Nothing can get in and nothing can get out. Your job is to perform a calculation on three numbers that will be provided to you when your effort is needed. Until then you wait.*
>
> *At some point during your time in the locked room three people come to the window. Each of them is holding a large flash card with a number on it. You do not know how they came up with the values, nor do you care. You write them down, in the order that they are presented, and get to work.*
>
> *When you are done you write your result on another flash card. You take it the window and hold it up to the glass. Someone on the outside - you do not know whom nor do you know what they will do with the information - writes it down. You erase your four values - the three inputs and the one output - from your workspace and go back to waiting.*

In this description, which may seem like a scene out of *Tron*, the function is completely isolated from the rest of the program. The function cannot access the outside variables. Instead the function must wait until the calling function provides it with the values in the variable. At no point can the function read or write to the outside variables. This is known as *passing by value*.

5.8.1 Local Variables

When data is passed by value the variable that is created to accept the value of the calling function variable is a *local variable*. It is a new variable completely separate from the variable in the calling function. Instead of the original variable, it is a copy. Its scope and lifetime is determined by the function that is being called.

Local Variable
A local variable is a non-global variable that is declared within a function. It has a scope and lifetime that are the same as the function in which it was created.

The new variable within the called function can have a different variable name than the variable in the calling

function or it can have the same name. If you use the same variable name in the calling function as you do in the function that is being called, it is important to understand that there are now two separate variables with the same name but different scopes.

Note that the the variable **x** in the driver - whose value of 3 is passed to the function - is a different variable than the variable **x** that is created to accept the value from the calling function.

```
function driver( )
%   DRIVER driver( ) is the main or driver function

    % Enter data into the program
    x = input( 'Enter a value for x: ');
    %   Call a local function
    y = calculation_function(x);
    % Print x and y
    fprintf('Driver (x, y) = (%0.2f, %0.2f)\n', x, y);

end
function m = calculation_function(x)
%   CALCULATION_FUNCTION m = calculation_function(x) is
%   a local function that's purpose is to demonstrate
    that data
%   that is passed by value creates separate variables
    m = x.^2 - 3;
    % Change x
    x = x + 5;
    % Print x and y
    fprintf('Function (x, m) = (%0.2f, %0.2f)\n', x, m);

end
```

```
>> driver( )
Enter a value for x: 3
Function (x, m) = (8.00, 6.00)
Driver (x, y) = (3.00, 6.00)

>>
```

Figure 5.27: Demonstrating the separation of variables in pass by value

To avoid any confusion, use a different variable name in the function from which the data is passed than in the parameter list of the function definition..

Since the default is that functions use encapsulation as a means to provide data integrity, is it possible to override this and change the scope of a variable multiple, non-nested, functions? It is and requires the creation of *global variables*.

Warning
When passing data to a function you may use the same or different variable names in the parameter list as you do in the function that has been called. Since they are two distinct variables that just happen to share a name, any changes to the variable in the function that was called will not be carried back to the variable in the calling function.

5.8.2 Global Variables

The default of making variables local is important in supporting the goal of encapsulation. By keeping variables within their own function creates data integrity. A function that uses a variable that may have the same name as that in another function cannot change the value of the second variable.

Global Variables
A global variable is a variable whose value is retained in memory after the function returns control. The value can then be shared by several functions. Its scope is all of the functions in which is declared as global. Its lifetime is the life of the program.

But there are instances when relaxing the control created by encapsulation can be useful.

> *You are writing a program to calculate deflections in a beam. Regardless of the loading on the beam the modulus of elasticity and the area moment of inertia will remain the same throughout the program. All of the functions that use these parameters are fixed and must remain the same for the run. But they may change the next time that the program is run. To guarantee that all of the functions use the same value for the parameters you would like to make the variables that store the parameters available to several different local functions.*

The approach to the scenario described is to create *global variables* to handle the parameters. A global variable, also known as a *static variable* is created within a function by adding the modifier global in front of the variable name. By doing this the value of the variable is retained in memory after the function is done and has returned control to the calling function.

```
1 global var
2 var = value;
```

Figure 5.28: Syntax for Declaring a Global Variable

When using a global variable for the first time in a program you would have to set it to an initial value. But after that the value that is stored in memory will be used.

An example of using a global variable as a parameter in a set of local functions is shown in figure 5.29.

In this example, the deflection of a beam is to be calculated. The deflection is determined in part by the modulus of elasticity, E, for the material and the area moment of inertial, I, for the shape of the beam. These two values remain the same for the different calculations as long as the beam

remains the same. To ensure that the same value is used in each calculation they are made to be parameters and as such global variables.

The scope of a global variable is all of the functions which declare the variable as global. When a function that uses a global variable returns control the program cannot know if the variable will be needed again, so the value is retained in memory. As a result, the lifetime of a global variable is the lifetime of the entire program.

The retention of a global variable is one difference from a local variable. There is a second. A local variable must be initialized with a value before it can be used in an operation. A global variable does not. If a global variable is not initialized then it receives a default value of zero.

This provides a useful method for counting how often functions get called. You create a global variable counter in the different functions that you want to count the total function calls. Since you do not know which will be called first you simply declare the variable and it will be set to zero by default. Then in each function you add in increment to the variable.

Adding these two lines of code to a function will result in the variable **counter** being set to zero the first time that it is encountered regardless of the function in which it is declared. It will then immediately be incremented by one. Since the value is retained the value will increase by one each time any of the functions are called.

```
function driver( )
% DRIVER driver( ) is the main or driver function

    % Create global variables
    global E;
    global I;

    % Enter Dimensions and Load
    a = input('Width of cross section (m): ');
    L = input('Length of beam (m): ');
    W = input('Load on beam (pounds): ');

    % Set the parameters
    E = 29.0e06;    % Modulus
    I = a.^4 ./ 12; % Moment of Inertia

    %   maximum deflection
    dPoint = deflectionCenterLoad(W, L);
    dUniform = deflectionUnifLoad(W, L);
    fprintf('\nDeflection of a Beam\n');
    fprintf('Point Load: %10.2f in\n', dPoint);
    fprintf('Uniform Load: %8.2f in\n', dUniform);

end
function d = deflectionCenterLoad(w, length)
% DEFLECTIONCENTERLOAD d = deflectionCenterLoad(w, l)
% calculates the center deflection of a beam with a center point load

    % Global Parameters
    global E;   % Modulus
    global I;   % Moment of Inertia
    % Deflection
    d = w .* (l.^4) ./ (48.*E.*I.*length);

end
function d = deflectionUnifLoad(w, length)
% DEFLECTIONUNIFLOAD  d = deflectionUnifLoad(w, length)
% calculates the center deflection of a beam with a uniform load

    % Global Parameters
    global E;   % Modulus
    global I;   % Moment of Inertia
    w = W ./ length;
    % Deflection
    d = 5 .* w .* (length.^4) ./ (384.*E.*I);

end
```

```
>> driver( )
Cross sectional width (m):
Length of beam (m):
Total load on beam (pounds):

Center Deflection of a Beam
Point Load:  m
Uniform Load:  m
>>
```

Figure 5.29: Using beam deflection calculations to demonstrate using a global variable as a parameter

```
1 global counter; %  Set to zero the first time it is
      called
2 counter = counter + 1;
```

Figure 5.30: Creating a global variable counter

5.9 Summary

5.10 Self Test

1. What is top-down design?
2. What is a function?
3. What is a built-in function?
4. What is the difference between a local function and the named function?
5. What is the purpose of **lookfor**?
6. What does the command **help** do?

5.11 Projects

1. Write a program that calls the print_

Relational Operations, Comparisons, and Piecewise Functions

6

What is the difference between a calculator and computer? Many people erroneously think that a calculator and computer are the same - that the only difference is speed and size. But this is not true. But it does pose the question; *What is the difference?*

A calculator has four functions - it can add, subtract, multiply, and divide. In fact a purist might argue that this is really only a single function, addition, since subtraction is simply addition of a negative number, multiplication is simply repeated addition, and division is just repeated subtraction, but for our purposes we will stay with the four.

Calculator	Computer
Addition	Addition
Subtraction	Subtraction
Multiplication	Multiplication
Division	Division
	Comparison

Table 6.1: Comparisons of the function of a calculator and a computer

So what are the functions of a computer? Like a calculator, a computer can add, subtract, multiply, and divide. But a computer has an additional operation - it can *compare*.

That is it. The only functional difference between a calculator and a computer is that a computer can do comparisons. So what is a comparison and how can we use it?

We often think of comparisons as in comparing two or more objects. For example, compare an apple and an orange. We look at the different colors, the different sizes, masses, and shapes. But in a program a comparison is more simple. Is the apple larger than the orange? Are the apple and the orange the same color? These are the types of comparisons that a program can make. Simple comparisons that result in true or false, or yes or no.

In a way the ability of a computer is like a game of twenty questions. It limited to answering yes or no. The program cannot check how much larger two values are - just that one is larger than the other. Or it cannot determine - directly - how two items are different. Just that they are different.

The comparisons the computer are not absolute. You cannot ask how much larger one value is compared to another. Or how they compare to all of the other values. They are relative solely to each other. In this way the computer can respond that it is true that one value is larger than the other, but not by how much, or how the two compare to a third.

As a result of this relative aspect when it comes to comparing variables, the operations are called *relational*; thus *relational operations*.

6.1 Relational Operations

A computer's ability to compare is about as basic as is its ability to perform arithmetic. We call these operations *relational operations* because they return a value that tells us the relative position of the two values.

Matlab can perform six different comparisons, thus there are six different relational operators, shown in table 6.2. They can be thought of as returning a true or false result regarding the relative position of the two values to each other. The six comparison operators are less than, less than or equal, greater than, greater than or equal, equal, and finally not equal.

Table 6.2: Relational Operators

Operator	Action	Example
<	Less Than	a < b
<=	Less Than or Equal	a <= b
>	Greater Than	a > b
>=	Greater Than or Equal	a >= b
==	Equal	a == b
~=	not Equal	a ~= b

Relational operators are more limited than are the arithmetic operators. Recall that **+** will add two values and return the sum, thus there are an infinite number of possible results from the addition operator. But relational operators are *logical* - they return only false or true.

Each of the six relational operators perform their comparison and return a *truth value* - that is true or false. In Matlab the truth values are 0 if the comparison is false, and 1 if the comparison is true. Nothing more - just those two results.

If you values on the number line, relational operators tell us if one of the values is to the left of the second, is in the same position as the second, or is to the right of the second. That is all. It will not determine the magnitude of the distance between the points. The result of the comparison **1** < **2** is just as true as is **1** < **5000** with both returning the logical value 1.

When an expression involving a relational operator is evaluated the result can be true, in MatLab this is the integer value 1, or false, the integer value 0. There are no other possibilities. The 1 or 0 are known as *logical values* or also as *boolean* or *truth values*. In MatLab these values are integers. Since they are integers, the truth value 1 for true will act the same as the value 1 in any operation. As such they can be stored in a variable or used in an arithmetic operation. Similarly, the logical value 0 indicates false. It is the integer value 0 and can be used in arithmetic operations.

Logical Value
A logical value, also known as a truth value, is the result of a relational operation. There are only two logical values - **true** which is represented by the integer value 1, and **false** which is represented by the integer value 0. logical values are often called *Boolean* values.

```
>>  5 < 9

ans =

    logical

    1

>>  3 >= 8

ans =

    logical

    0

>>  4 == 36./9

ans =

    logical

    1

>>  x = 15*2;
>>  y = 75./3;
>>  x ~= y

ans =

    logical

    1
```

Figure 6.1: Calling a relational operations from the command line

6.2 Relational Operations as a Step Function

The values that are returned from a relational operation are always 0 for *false* and 1 for *true*. The 0 and 1 are not symbolic - or enumerated - they are the actual numerical values. As such they can be used numerically.

If you multiple by a *false* logical value you will be multiplying by 0 and the result will be 0. Further, if you multiply by a logical *true* you are multiplying by 1 and the value will remain the same. Similarly you can change a false to a true or a true to a false by simply subtracting the value from 1. After all, $1 - 0 = 1$ and $1 - 1 = 0$.

Note
There is a selection structure that we will present later in this chapter that is known as a *switch* so while we often think of the current application operating as an *off - on switch*, to avoid confusion

6.2.1 Programming a Piecewise Function

The use of the numerical aspect of logical values enables us to use them as a step - that is a way for a function to jump from one value, or one curve, to another.

We can demonstrate the application with an example.

Example A company manufactures widgets and sells them at a price p. To simplify purchasing the widgets the sales team would like a function that can be used to calculate the cost. If you purchase q items at a price p then the cost is $p \cdot q$. But there is a catch. If you just say that $c(q) = pq$, what happens if the user enters a negative value for q?

The simple function would return a negative value. Instead we need to create a piecewise function for the cost such that it is 0 if $q < 0$ and $p \cdot q$ if $q \geq 0$. This piecewise function is

$$c(q) = \begin{cases} 0 & \text{if } q < 0 \\ pq & \text{otherwise} \end{cases}$$

We can program this using the relational operation **q > 0**. Writing it as a local function

```
function c = cost(q, p)
%   COST c = cost(q, p) calculates the cost of
    purchasing
%   q items at a price of p

    % Calculate cost
    c = (p .* q).*(q >= 0);

end
```

```
>> price = 10.0;
>> quant = -5
>> c = cost(price, quant)
c =
    0.00000

>> quant = 8;
>> c = cost(price, quant)
c =
    80.00000
>>
```

Figure 6.2: Piecewise cost function

The local function in figure 6.2 demonstrates the code for a piecewise function with a single step. It is possible to have

two steps, or three, or as many as you need. But to do this, you need to turn the previous step off - that is return it to zero - before turning the next step on. We do this with a second relational operation; this time less than.

Example The same company from the previous example provides price breaks for large purchases. If you purchase less than ten of the items they are $8.00 a piece. But if you buy ten or more the price drops to $5.00 each but with an additional fixed fee of $30. This fixed fee keeps the cost function continuous.

The piecewise function is now

$$c(q) = \begin{cases} 0 & q < 0 \\ 8q & 10 \leq q < 10 \\ 5q + 30 & q \geq 10 \end{cases}$$

In the same manner as the previous example we start by turning the function on at zero, but now we have to step back to zero when $q = 10$. We do this with a second relational operator multiplied to the first. Once that price break is met we add a second function on to the first, but this one has a second step to one when $q = 10$

We can extend this function to as many price steps as you need. Each one begins with a relational test at its starting point and, with the exception of the final price, stops with a second test at the beginning of the following price.

The price break model shows how relational operators can be used to create discrete steps. Another application is in modeling piecewise functions.

6.2.2 Heaviside Function

Piecewise functions are ubiquitous in engineering. Anytime that we need to start or stop one function and replace it with another we have a piecewise function.

Example A piecewise function can be used to model the activation of a piece of machinery. Let the system consist of starting a flywheel at time $t = 10$ and having it accelerate to a operational speed in five seconds. In this function we will need the output to be at zero until it is started. At that point it will accelerate using a linear function for the next five

```
function c = cost(q)
%   COST c = cost(q) calculates the cost of purchasing
%   q items at a price of 8 until q reaches 10 where the
%   price drops to 5

    % Calculate cost
    c = (8 .* q).*(q >= 0).*(q < 10) + (5.*q + 30).*(q
    >=10);

end
```

```
>> quant = -5
>> c = cost(quant)
c =
    0.00000

>> quant = 4;
>> c = cost(quant)
c =
    32.00000
>> quant = 15;
>> c = cost(quant)
c =
    105.00000
>>
```

Figure 6.3: Piecewise cost function with two price breaks

seconds. After five seconds it stays at that constant rotational speed. A piecewise function to model this would be

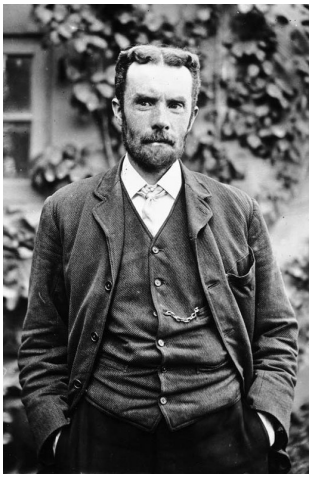
$$c(t) = \begin{cases} 0 & t < 10 \\ 3(t-10) & 10 \leq t < 15 \\ 15 & t \geq 15 \end{cases}$$

We could use relational operators to act as the switch - turning on the first function, then switching it off and starting the second - but this type of model is so common that it is worth developing an alternative approach to model piecewise functions; the Heaviside function.

The Heaviside function is the following unit step functionas shown in figure 6.5.

$$H(t) = \begin{cases} 0 & t < 0 \\ 1 & t \geq 0 \end{cases}$$

The value of this in modeling piecewise functions is that it is a specific relational operator so it works exactly the same. As



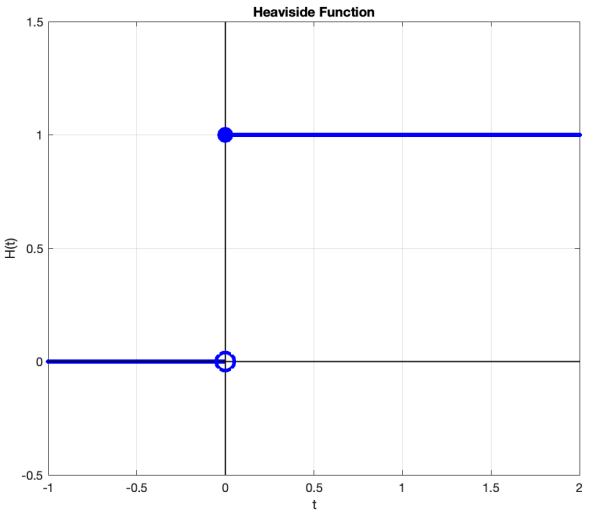


Figure 6.5: Plot of the Heaviside function

before the return values are 0 and 1. They function the same as do the relational operations earlier.

Since the step occurs at 0, if we want to switch a function from 0 to 1 at $t = 0$ we just have to multiply the function by $H(t)$. Using the Heaviside function, the ramp function,

$$f(t) = \begin{cases} 0 & t < 0 \\ t & t \geq 0 \end{cases}$$

can be modeled as

$$f(t) = t \cdot H(t) \tag{6.1}$$

If the input to this function is negative, the return is 0. But if t is zero or above, the return is t. We can show this by looking at some values (table 6.3)

Table 6.3: Discrete values of the ramp function in equation 6.1

t	$f(t) = t \cdot H(t)$
−2	$-2 \cdot 0 = 0$
−1	$-1 \cdot 0 = 0$
0	$0 \cdot 1 = 0$
1	$1 \cdot 1 = 1$
2	$2 \cdot 1 = 2$

The Heaviside function is already built in to the MatLab library but creating a local version of it provides an increased

understanding of how it works and why. The body of the function is only a single line - a relational operation to check if the input value is greater than or equal to zero.

```
function h = H(t)
%   H h = H(t) implements the Heaviside functon
%   return 0 if the input is negative and 1 if it
%   greater than or equal to 0

    % Test the input
    h = (t >= 0.0);

end
```

```
>> x= -5
>> s = H(x)
s =
    0.00000

>> x = 4;
>> s = H(x)
s =
    1.00000
>>
```

Figure 6.6: Heaviside Function

The Heaviside function can be called as **H(t)** if the breakpoint of the piecewise function is at $t = 0$, but what if it is not? In algebra we learned that to move a function horizontally, you subtract the amount of the translation from the input. Thus to move the function $a > 0$ units in the positive direction you subtract a from the input. If the function is to moved in the negative direction then you subtract $-a$, $a > 0$, from the input. This is the same with the Heaviside function. If the breakpoint is moved from zero to a, then you subtract a from the input. Thus **H(t - a)** is the unit step function except now the breakpoint is at $t = a$ instead of $t = 0$

Note
Recall from Algebra that given a function $f(x)$, the function $f(x - a)$ is the same function shifted a units in the positive direction.

Example Write an anonymous function for the piecewise continuous function

$$f(t) = \begin{cases} 0 & t < 3 \\ t^2 - 9 & t \geq 3 \end{cases}$$

The piecewise function can be created using the standard form of an anonymous function where the expression is the second case in the piecewise function - the first is not needed since it will already be zero.

At this point we have a means of starting a piecewise

```
>> f = @(t) (t.^2 - 9) .* H(t - 3);
>> y = f(2)
y =
    0.00000

>> y = f(4)
y =
    7

>> y = f(8)
y =
    55

>>
```

Figure 6.7: Anonymous Piecewise Function

function, but what about stopping it? Since the Heaviside function is a *step up* from 0 to 1, switching from on to off is the same as stepping down from 1 to 0. This requires a step function that is equal to 1 before the breakpoint and switches down to 0 at that point. The step down function can be created by reversing the Heaviside function. Since the reverse step will be 1 if $H(t) = 0$ and 0 if $H(t) = 1$ the new function is just $1 - H(t)$.

Table 6.4: Heaviside as a Step Down Function

$H(t)$	$1 - H(t)$
0	$1 - 0 = 1$
1	$1 - 1 = 0$

In starting a function at a first breakpoint, a, and then stopping it at a second breakpoint, b, we multiple the two functions together. A simple application is to create a function that steps from 0 to 1 at a and then steps back down to 0 at b.This new function

$$f(t) = \begin{cases} 0 & t < a \\ 1 & a \leq t < b \\ 0 & t \geq b \end{cases}$$

can be modeled as

$$f(t) = H(t - a) \cdot (1 - H(t - b)) \tag{6.2}$$

Table 6.5 shows how this new function steps from 0 to 1 and back to 0.

Returning to the example that opened the section, to model a

t	$H(t-a)\cdot 1 - H(t-b)$
$t < a$	$0\cdot(1-0) = 0$
$a \le t < b$	$1\cdot(1-0) = 1$
$t \ge b$	$1\cdot(1-1) = 0$

Table 6.5: Heaviside as a Step Down Function

piecewise function you multiply the function by the product of the step up and the step down forms of the Heaviside function. If there are two - or more - functions you add the additional functions on with their own breakpoints.

If the two functions are $f(t)$ and $g(t)$ and there are breakpoints at a and at b, the anonymous function becomes

```
p = @(t) f(t - a).*H(t - a).*(1 - H(t - b)) + g(t - b).*H(t - b);
```

Returning to the original example

$$c(t) = \begin{cases} 0 & t < 10 \\ 3(t-10) & 10 \le t < 15 \\ 15 & t \ge 15 \end{cases}$$

can be modeled with

```
>> p = @(t) 3.*(t-10).*H(t-10).*(1-H(t-15)) + 15.*H(t
    -15);
>> y = p(2)
y =
    0.00000

>> y = f(12)
y =
    6.00000

>> y = f(16)
y =
    15.00000

>>
```

Figure 6.8: Anonymous Piecewise Function

6.3 Boolean Expressions

6.3.1 Order of Precedence

Recall that the arithmetic operators, addition, subtraction, multiplication, division, and exponentiation, have a specific order of precedence when being used - **PEMDAS**. Similarly, relational operations have an order of precedence as well. In this case the order is **less than** (<), then *less than or equal* (<=), *greater than* (>), *greater than or equal* (>=), and finally *equal* (==), and *not equal* (~=). In the same way as arithmetic, if the operations are at the same level, for example if a comparison contains two less than relational operations it performs the comparisons from left to right.

When entered one at a time as shown in figure 6.9, the results of the relational operation are exactly as expected. Byt when written as a single, multiple comparison, the results are not. In fact, no matter what value of **x** is entered into the relational operations the result will always be *true*, 1. It is possible to create a similar sequential relational comparison in which the result is always *false*, 0. The reason is that since the return values of a relational operation are numerical the first comparison will always be 0 or 1. The second comparison will then use the 0 or 1 when performing the second comparison.

```
>>  x = 2;
>> s = (3 < x)

s =
    0

>> s = (x < 5)

s =
    1

>>  s = (3 < x < 5);

s =
    1

```

Figure 6.9: Result of Multiple Comparisons

The unexpected logical value is due to the comparisons being made left to right. By adding in parentheses where they are implied, you can see the actual results one at a time. The first comparison returns 0 since it is false, after all two is not greater than three. But the next comparison is not with the

value stored in the variable, it is instead a comparison to the logical value from the previous comparison. with the zero and that one is true, thus 1.

All types of multiple comparisons will be addressed using Boolean Expressions in section 6.3, but the example shown above is special case that we will call a *step*.

Example
You can see how multiple comparison returns an unexpected result by looking at the comparisons one step at a time

$$\begin{aligned} s &= (3 < 2) < 5 \\ &= 0 < 5 \\ &= 1 \end{aligned}$$

Even though the statement as written is clearly false, the program returns it as true.

Selection Structures and Branching

7

Dorothy : Now which way do we go?
Scarecrow : Pardon me. That way is a very nice way.
Dorothy : Who said that?... Don't be silly, Toto. Scarecrows don't talk.
Scarecrow : It's pleasant down that way, too.
Dorothy : That's funny. Wasn't he pointing the other way?
Scarecrow : Of course, people do go both ways!

Wizard of Oz (1939)

```
function x = get_data_range(prompt, m, M)
%   GET_DATA_RANGE   x = get_data_range(prompt, m, M)  prompts the
%   user to enter a value and then error checks that is  between a
%   minimum, m, and a maximum, M
%

    %   Prompt the user to enter a value
    x = input(prompt);
    %   Check if the value is less than the minimum
    if(x < a)
        %   Value is too small so print an error message  and exit
        error('Value %0.2f is below the minimum of %0.2f ', x, a);
    end

    %   Check if the value is above the maximum
    if(x > b)
        %   Value is too large so print an error message  and exit
        error('Value %0.2f is above the maximum of %0.2f ', x, b);
    end

end
```

Figure 7.1: Using a simple if to error check an input

Recursion 8

Lather - Rinse - Repeat
Instructions on the back of a shampoo bottle

Recall that the purpose of a function is to simplify a program. Instead of having to rewrite code multiple times we simply write a function to perform some task for us. Then whenever we want that action we call the function.

There is a process that can be used in functions that allow a function to called from within itself. This process is known as *recursion*.

Recursion provides a technique for repeating a function by having the function call itself.

Recursion
Recursion is a problem solving technique in which the solution is found by applying the same technique to smaller instances of the same problem.

8.1 What is Recursion?

While few of us have ever thought of it as such, recursion is how we run our lives. By performing the same tasks over and over again - not a specific number of times, but when we need to.

> *Think of walking out of a building. You do not say "I will take fifteen steps and then turn left." No, you take one step towards your destination. If you are not at your destination you do it again - take a step towards your destination. You continue this until you arrive. This is recursion.*
>
> *Another common example of recursion is the directions on a shampoo bottle - Lather - Rinse - Repeat. You can see this in the flow chart in figure 8.1. While you might think of this as an example of a repetition structure it is open ended - something that was made for recursion. It does not say Do this twice: Lather - Rinse. Instead, it is actually a selection. It becomes so when we added the logical condition Does your hair need to be washed?*

Recursion is a combination of a selection in the form of either a *simple if* or an *if-else* and a function call. But the function call is what so often bothers us – it is a function call to the same function which is currently active. This is where the definition of recursion comes in – recursion is a technique

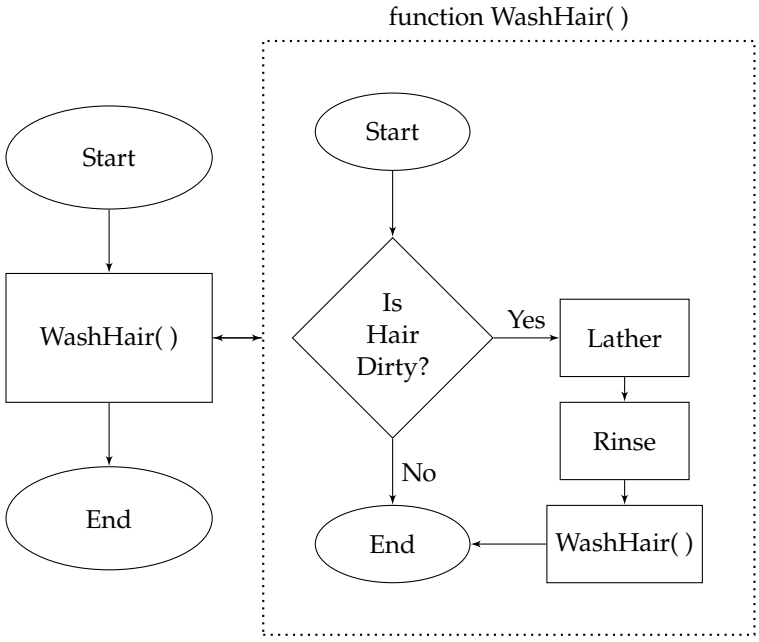


Figure 8.1: Flow Chart of a Recursive Function Call

that solves a problem by repeating the same solution on a smaller piece of the task.

8.2 Implementing Recursion

We used an example from our daily routine to introduce the concept of recursion, but why use it in a program? Because just like the daily routine example solving problems with recursion is a natural method that we use to solve many problems.

Many recursive applications are computational. We will see examples for calculating factorials, and summations, and combinations. But others are not. Instead they perform a task, and if it is not completed – or completed incorrectly – then the recursive continues the process.

Examples of the non-computational use of recursion are often more direct. After all they are techniques that appear more natural because it might be how we would do the operation if we were not on the computer. The example that we will use to introduce the technique is error checking an input. But it can also be used for searching through data looking for a specific value or item, or it can be used to sort data – these last two will have to wait until we introduce vectors and lists. But error checking is a simple starting point for creating recursive functions.

8.2.1 Error Checking

Error checking is one of the most important processes in programming. The purpose is simple - counter the old saying of *GIGO* or *Garbage In - Garbage Out*. The recursive approach for error checking combines a means of entering the data into the program with the error check.

Error checking is also a natural recursive action. Imagine that you are at a deli counter. Your number gets called and you ask for a half pound of sliced cheddar. The deli clerk goes back and returns with a pound of provolone. They made an error, so you tell them that it is incorrect. Since the chances are that the clerk simply misheard you the first time, or became distracted, or simply forgot. Regardless, the most direct approach to remedy this is for you to ask for deli clerk for the cheddar again - textbook recursion!

This error checking function, figure 8.2, does just this. It starts with printing a prompt to the user to enter the data. It then checks if it is within the specified range of values using the relational operator approach from chapter 7. This is known as

```
function x = get_data_range(prompt, a, b)
% GET_DATA_RANGE x = get_data_range(prompt, a, b) is a
% recursive function that error checks the entered data. It
% checks if the value is between a and b. If it is not then
% it prints a warning and calls the get_data_range function
% recursively.
% The stopping condition uses a simple if, not an if - else.
%

    % Enter the data using a call to input
    % with the string prompt
    x = input(prompt);

    % Create two logical variables to check range
    s = (x >= a); % These could be replaced with
    t = (x > b);  % calls to the Heaviside function


    %  Stopping condition
    %  if it fails print warning and call get_data_range
    if(s.*(1-t) == 0)
         warning('%g outside of %0.3g to %0.3g', x,a,b);
         x = get_data_range(prompt, a, b);
    end

end
```

Figure 8.2: MatLab code for Error Checking

warning
Similar to **error**, MatLab has a function called **warning** that prints a message to the user, but instead of exiting the program it then allows it to continues.
Syntax:
warning(prompt);

the *stopping condition* or *stopping case*. If the check fails, the function prints a warning and then calls the function again. if the range check passes then the function returns the entered value to the calling function - the *stopping*, or *base case*.

This function could be adapted with other error checking in addition to - or instead of - the range of the input. You could check the type of value that was entered. For example that the value is an integer, or a boolean value. Or you could change the input to accept a string and then check that the text of the string contains a specific substring.

The important concept here is how the recursion worked. It is

functionally similar to the error checking function that we created in chapter 7. But that function was a one off. The user was given the chance to enter the data. Once they did it error checked in the exact same way. But now if the user made a mistake in data entry it just warns them and lets them try again. Our first step into repetition.

8.2.2 Using recursion for calculations

Recursion as a tool for performing actions is valuable, but we also want to perform calculations. And recursive approaches to computation can at times be the difference between being able to complete a calculation or not.

The reason for a recursive approach is two fold. The first is that whatever the calculation is it is naturally recursive. We will see this in both the factorial calculation and the fibonacci series.

The second reason is that there are calculations that when written in their common form would overwhelm the constraints of the computer. This is the case in calculating combinations.

Factorials are a common calculation in combinatorics and in probability. The formula is well known

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

While many people would calculate the value starting at one, it is more direct to start at n just the way the formula is always presented. The reason is that this formula is naturally recursive.

The formula for $(n-1)!$ is

$$(n-1)! = (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

Substituting this into the original formula for the factorial results in the recursive form

$$\begin{aligned} n! &= n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \\ &= n \cdot (n-1)! \end{aligned} \tag{8.1}$$

A recursive approach to calculating 4! from the top down

$$\begin{aligned} 4! &= 4 \cdot 3! \\ 3! &= 3 \cdot 2! \\ 2! &= 2 \cdot 1! \\ 1! &= 1 \cdot 0! \\ 0! &= 1 \end{aligned}$$

and then from the bottom back up

This recursive form is easily explained as the same calculation being done but on a different value. In other

words, a factorial is actually the product of a single value and a different factorial.

Using this recursive approach if someone asks you what is four factorial, you answer "Easy - its four times three factorial!" And of course three factorial is three times two factorial. You just keep calling the factorial function on smaller and smaller values of n.

Of course you will need to stop at some point. You do this by recalling that 0! = 1. Thus when you get to 0! you replace it with the known value of 1 and start working back up through the calculations.

In this example knowing to stop at 0! is called the *Stopping Condition* or *Stopping Criteria* while the value 0! = 1 is the *Base Case*.

The factorial function becomes a nature form of using this recursive formula. As such we can now write the formula as a function with which we can calculate any value of n as long as n is a non-negative integer.

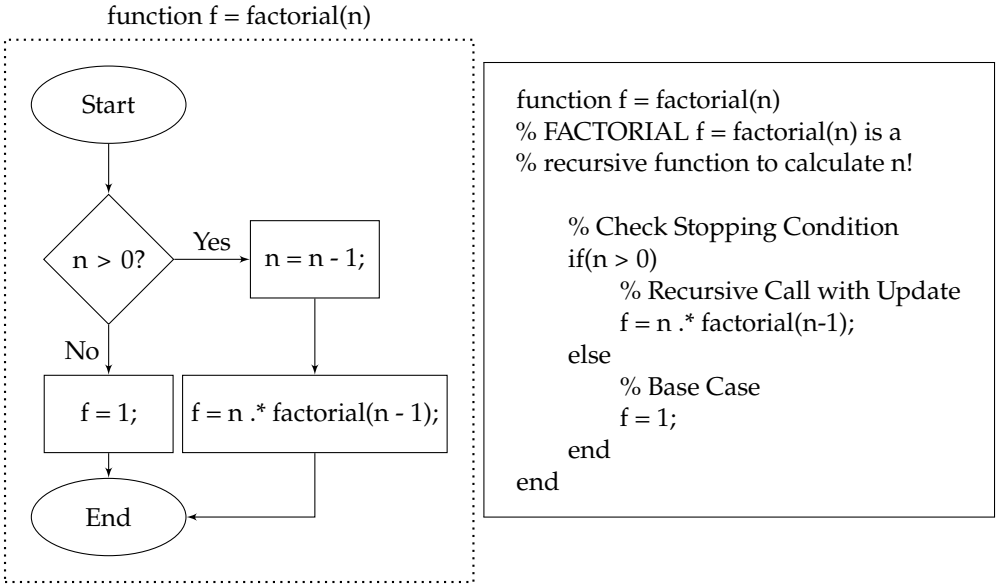


Figure 8.3: Flow Chart and Matlab Code for a Recursive Factorial Function

8.3 Theory of Recursion

While it may appear that recursion is repetition, it is actually a selection structure. The program calls a function. Within

the function it evaluates whether or not some stopping condition is satisfied. Depending upon that result the function will either perform some operation(s) then call the function with an updated set of input parameters; or it sets a base case and returns control back to the driver. No repetition - just a selection in the form of an *if - else* structure.

8.3.1 Three Rules of Recursion

A common question about recursion is "how do we know that it works?" The first thing to note is that recursion has a simple process - *you call the function - the function makes an update to itself - the function then calls it itself.* If we implement this correctly then it should work - but will it always work?

Stopping Condition
Stopping Condition is the test whether the recursive function either update and calls the function again, or returns the base case.

This actually has a very simple answer. A recursive function will work as long as it satisfies the three rules of recursion.

1. The base case must behave correctly.
2. The stopping condition must result in a change in the inputs and move toward the base case.
3. The stopping condition must call the function.

Base Case
Base Case is the branch of the stopping condition that does not call the recursive function. The base case often returns a final value although it may also just return control of the function.

The flow chart in figure 8.4 shows both the general form of a recursive function and the three rules.

While we will present it without proof, the three rules are both necessary and sufficient. This means that as long as our recursive function satisfies these criteria the recursion will work, and return the correct result.

8.3.2 Stack Memory

Recursive functions are a simple but powerful method for a program. Nevertheless there is a possible issue with their use - memory.

When a computer program runs the computer allocates a block of memory for the program. This memory is called the *stack*.

Stack Memory
Stack memory is a region of memory that stores the local variables for a function.

Each time that a function is called the stack stores the local variables for the function that are created. These variable remain in the stack as long as the variable is active. The stack operates as last in - first out (LIFO). This means that each time a function is called the memory needed for that function's variables are pushed on to the top of the stack.

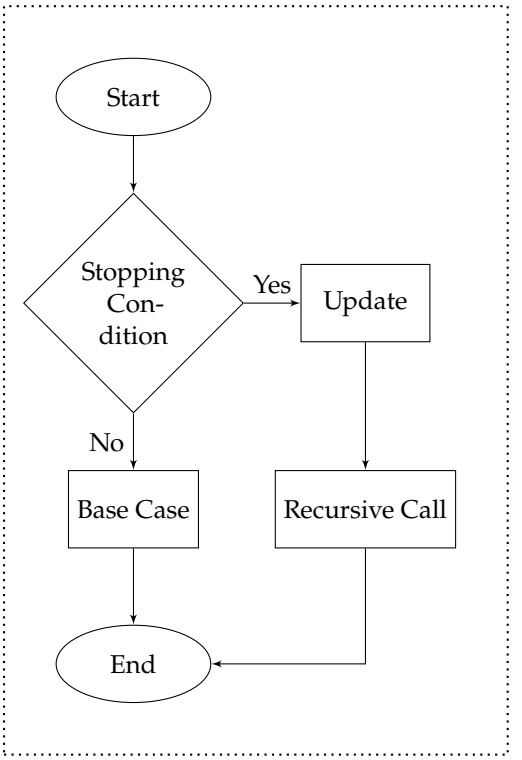


Figure 8.4: Flow Chart for a General Recursive Function

When that function exits, that memory is released and returned to be used for the next function that is called. As long as a function is active the stack memory for other functions are not available.

LIFO
LIFO is a queueing nomenclature for *Last In - First Out*. It indicates that when an item enters the stack it goes on the top pushing all others down. It will also be the first one to be *popped* or removed from the stack.

An analogy of the stack is to imagine a function is a sheet of paper. Each time that a function is called a new sheet of paper is added - *pushed* - onto the top of the pile - *stack* - making the other sheets of paper unavailable. As long as the function is active all memory activities take place on that top piece of paper. When function returns control the sheet of paper is removed and the paper - *memory*) -below it moves to the top and becomes active.

Stack Overflow
Stack overflow occurs when a function is called after all of the memory available for the stack has been allocated. It is a type of run-time error.

In recursion, each time that the function calls itself an additional block of memory is allocated for the function. While not normally an issue, the memory available for the stack is finite. This means that if the recursive function continues to call itself many times it is possible to allocate all of the memory that has been reserved for the stack. If so the next time that the function is called there will not be memory available for it and the program will crash. This is a type of

run-time error known as a *stack overflow error*.

Stack overflow is commonly a result of *infinite recursion*. A misnomer because the function has not called itself an infinite number of times, but it is does occur when the recursive function as called itself so many times that there is no longer any memory remaining in the stack.

Infinite Recursion
Infinite Recursion is a run-time error resulting in a stack overflow. It is usually a result of an incorrect stopping condition.

The most common reason for infinite recursion is an incorrect stopping condition. For example, if the function should update as long as $n > 0$ but the update adds one to n instead of subtracting the stopping case will never be reached. Eventually all of the memory in the stack will be allocated and the stack overflow error will occur.

8.3.3 Direct and Indirect Recursion

Using recursion to calculate the factorial is an example of direct recursion. Within the recursive function the same function is called again. But what if the recursive function calls another function which in turn calls the original function? This is still recursion but it is no longer direct, but is now indirect.

Recall that the factorial function calls itself recursively (figure 8.3). This is direct recursion. So how would this change for indirect recursion?

In indirect recursion the recursive function calls a different function which in turn calls the original function. This makes it appear that individually neither function is a recursive function - but when viewed together it is.

Indirect Recursion
Indirect recursion is when the recursive function calls a different function which in turn calls the recursive function.

An interesting example of this is the indirect recursive function to determine if a number is even or odd. In this example the function first checks to see if the entered value is zero. Since zero is not odd it will by default return 0 for false.

If the value is not zero it then moves the **isEven** function for $n - 1$. If the value of n was originally equal to one then the new value is 0 and the function returns 1 for true - the current value is even and thus the original value was odd.

As before, the benefit of the recursive function is its simplicity in coding. Figure 8.6 shows the indirect recursion as MatLab code. If you had only seen one of these functions you would not have been able to identify it as a recursive function, but by looking at both together you should recognize that it is recursive and an example of indirect recursion as well.

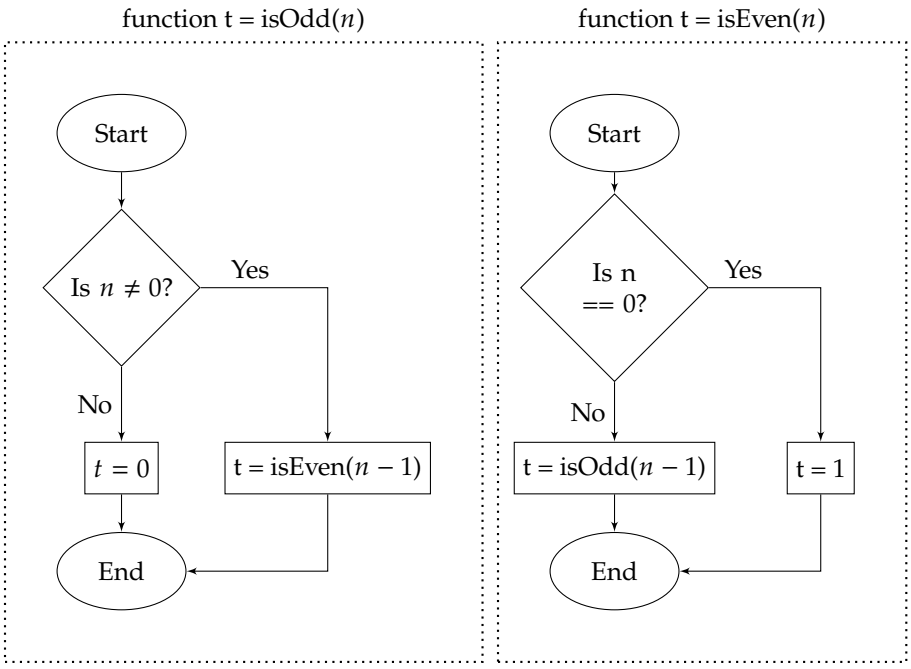


Figure 8.5: Flowchart demonstrating indirect recursion

```
function f =
 isOdd(n)
 % ISODD t = isOdd(n)
 % determines if n is
 odd

 % Stopping Condition
 % 0 is not odd
 if(n ~= 0)
 % Recursive Call
 % with Update
 t = isEven(n-1);
 else
 % Base Case
 t = 0;
 end
 end
```

```
function f =
 isEven(n)
 % ISEVEN t =
 isEven(n)
 % determines if n is
 even

 % Stopping Condition
 % 0 is not even nor
 odd
 if(n == 0)
 % Base Case
 t = 1;
 else
 % Recursive Call
 % with Update
 t = isOdd(n-1);
 end
 end
```

Figure 8.6: MatLab code for indirect recursion

8.3.4 Complexity

As we have with all implementations we are interested in the computational complexity of recursive functions. While most recursive applications are simple and fast $O(n)$, it is possible to have a recursive algorithm that is beyond the acceptable complexity levels. The issue can be determined by whether the function implements single recursion or multiple recursion.

Single Recursion

Single recursion is just what it says. The recursive function - either direct or indirect - makes a single recursive function call. Each example that we have already seen is an example of single recursion. The complexity of single recursion can be determined in a function tree for the recursive factorial function as shown in figure 8.7. This example of single recursion is $O(n)$.

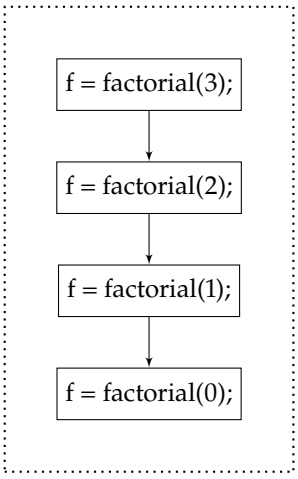


Figure 8.7: Hierarchical diagram of single recursion

While adjusting the input by subtraction is commonly $O(n)$, it is possible to have a single recursive function that is better than that. An example code is shown in figure **??**. In this example the update is divided by two each time

Multiple Recursion

Multiple recursion is different. In multiple recursion the function makes more than one recursive function call each time. An example of multiple recursion is calculating the Fibonacci sequence.

The fibonacci sequence goes back to the thirteenth century. It is a sequence that is commonly found in nature. Its definition

Fibonacci
Fibonacci was a thirteenth century mathematician from the Republic of Pisa. Also known as Leonardo of Pisa he is often thought to be the most influential western mathematician of the middle ages. He most often remembered for the sequence named after him.

is recursive - each value is determined by the sum of the two values that come right before it.

$$\begin{aligned} F(n) &= F(n-1) + F(n-2) \\ F(0) &= 0 \\ F(1) &= 1 \end{aligned} \tag{8.2}$$

The sequence in equation 8.2 is clearly recursive. Each term is calculated by adding the two that come before it. To calculate the sequence

The The hierarchical diagram in figure 8.8 shows that the

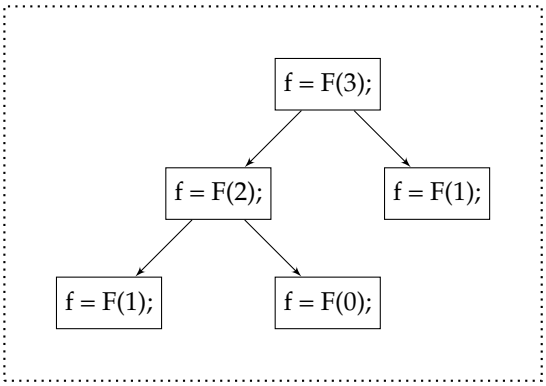


Figure 8.8: Hierarchical diagram of multiple recursion

8.4 Applications of Recursion

Recursion can be used anytime that it is necessary to repeat a process. After all, recursion is a type of repetition structure. We will see that there are additional repetitions structures which we will call loops, There are actually programming languages that do not have loops but rely solely on recursion. While MatLab is not one of them, it is often possible to use recursion instead of a loop when a repetition is necessary.

There are many useful applications for recursion. They range from functions that perform computations to those that provide support to other actions. We already looked at error checking an input, but there are also applications in searching and sorting data.

8.4.1 Computation

When it comes to computation, many recursive functions are designed to simplify some repetitive operation. We have already seen the example of calculating a factorial. Recall that by its definition it is a recursive function, but we will see that it can be calculated using a loop, A similar example is the calculation of permutations and combinations. But the loop approach to these calculations has a different problem – the magnitude of the numbers. As a result, recursion becomes a necessity.

Both permutations and combinations belong to a class of functions used in *combinatorics*. Combinatoric calculations are a basis of theoretical probability. As mentioned, the two primary combinatoric calculations are permutations and combinations. Each of them determines the number of selections that can be made from a set of n items when you are picking k at a time.

The number of permutations of n items selected k at a time is the number of ordered sets of k that can be formed from the n.

$$\begin{aligned} P(n,k) &= n\,(n-1)\,(n-2)\cdots(n-k+1) \\ &= n!/(n-k)! \end{aligned} \tag{8.3}$$

Combinations are similar to permutations, but the selected subsets are not ordered (the order does not matter). It is calculated by first calculating the permutations and then removing all the repeated selections. The calculation is product and quotient of factorials.

$$\begin{aligned} C(n,k) &= P(n,k)/P(k,k) \\ &= \tfrac{n!}{(n-k)!k!} \end{aligned} \tag{8.4}$$

Permutations can be calculated using two factorials and combinations with three. The three factorials would at first make this appear to be an example of multiple recursion. But if you wrote the function using the three factorial functions it would actually works out to be singular recursion three times. As such it would still have complexity of $O(n)$. But programming the function in this form is still not

recommended. The issue is the magnitude of the calculations. To show this, try to calculate $C(100, 50)$ using factorials.

The alternative is to simplify the calculation to a single recursive function. But how?

Calculating combinations and permutations have been done for hundreds of years. When doing this by hand the factorial approach is untenable. But if you write out the factorial form you may see a simplification; one of the factors in the denominator will always cancel out. To see this try $C(15, 9)$

$$\begin{aligned} C(15,6) &= \tfrac{15\cdot14\cdot13\cdots12\cdot11\cdot10}{(9\cdot8\cdot7\cdots2\cdot1)(9\cdot8\cdots2\cdot1)} \\ &= \tfrac{(15\cdot14\cdots8\cdot7)(6!)}{(9\cdot8\cdot7\cdots2\cdot1)(6!)} \\ &= \left(\tfrac{15}{6}\right)\left(\tfrac{14}{5}\right)\left(\tfrac{13}{4}\right)\cdots\left(\tfrac{9}{1}\right) \end{aligned}$$

When you look at this calculation, what may jump out at you is the recursive nature of it. It is nearly the same as the factorial calculation. By grouping we see a new relationship

$$\begin{aligned} C(15,9) &= \left(\tfrac{15}{6}\right)\left[\left(\tfrac{14}{5}\right)\left(\tfrac{13}{4}\right)\cdots\left(\tfrac{10}{1}\right)\right] \\ &= \left(\tfrac{15}{6}\right)(C\,(14,5)) \end{aligned}$$

The sample shows how to identify the recursive relationship. We can generalize it to

$$C(n,k) = \left(\frac{n}{k}\right) C\,(n-1, k-1) \tag{8.5}$$

Writing the recursive relationship for combinations as a function is shown in figure 8.9.

```
function c = comb(n, k)
% COMB c = comb(n, k) is a recursive function that calculates
% the number of combinations of n items taken k at a time
%

    %  Check if k > 0
    if(k > 0)
        %  Update
        c = (n ./ k) .* comb(n-1, k-1);
    else
        %  Base Case
        c = 1;
    end

end
```

Figure 8.9: MatLab code for Calculating Combinations

Repetition Structures 9

There is a construct in computer programming called 'the infinite loop' which enables a computer to do what no other physical machine can do - to operate in perpetuity without tiring. In the same way it doesn't know exhaustion, it doesn't know when it's wrong and it can keep doing the wrong thing over and over without tiring.
John Maeda - American Designer

Repetition occurs over and over - no pun intended - in many fields. Some literary examples are

> "Let it snow, Let it snow, Let it snow." - Sammy Cahn

> "Miles to go before I sleep, and miles to go before I sleep.
> Robert Frost

> "But I would walk 500 miles
> And I would walk 500 more
> Just to be the man who walked a thousand miles
> To fall down at your door" - The Proclaimers

In literature repetition is the act of repeating the same line a set number of times or until some other action causes it to stop. But how does this apply to computation?

Repetition Structure
A repetition structure is a block of instructions that are repeated sequentially as long as some condition is met.

9.1 Loops and Repetition

I worked with a man who once told me "a computer does not do anything that any of us cannot also do. It just does it over and over without complaining." This is repetition.

Repetition in a program is the act of repeating a block code either a fixed number of times, or until some condition is met, or once for each item in some predetermined list. The first two are examples of convergence, while the third is better known as iteration.

Loop
A loop is a sequence of instructions in a program that can be executed repetitively either a fixed number of times or until certain conditions are satisfied.

Repetition structures are more commonly known as *loops*. In their most basic form a loop is a block of code that once begun will potentially run multiple times exclusive of the rest of the program. Each time that the block of code runs is known as a *pass*.

Pass
A pass is a single run through the block of instructions in a repetition structure.

An historical example of repetition is actually a story of avoiding repetition.

> *The story - perhaps apocryphal - is of a nine year old Karl Friedrich Gauss. A version of this is that his teacher as punishment to an unruly class directed the students in the class to add the numbers from one to one hundred most likely assuming that the task would take the class at at least an hour.*
> *While the class worked away on the exercise, Karl simply sat idle for a few minutes, then picked up his pencil and wrote a number on his paper, and put the pencil back down.*
> *The teacher was indignant at the impertinence of the student ignoring his assignment so he picked up the paper and saw the answer written on it* – 5050.
> *When confronted with his insolence, Gauss pointed out that he was able to simplify the task of repeated addition with a simple product.*

$$\begin{aligned} S &= 1 + 2 + 3 + \cdots + 98 + 99 + 100 \\ &= (1 + 100) + (2 + 99) + (3 + 98) + \cdots + (50 + 51) \\ &= 50 \cdot 101 \\ &= 5050 \end{aligned} \tag{9.1}$$

> *In this case Gauss was able to simplify a repetitive process in a way that was not obvious at the start. Instead of requiring a hundred passes through a loop, he was able to do it in two steps - a single addition followed by multiplication.*

The Gauss example is a mathematical example of avoiding repetition. Using repetition Gauss would have started with 1. Added 2 to it, then 3 and so on until he had reached 100. The repetition was not in the values but in the process. He repeated the same process of adding a value to the sum until some goal had been reached.

Repetition structures - loops - come in two types; convergence loops and iterative loops. What is the difference between the two and how are they the same?

9.2 Convergence Loops

A convergence loop is one in which the number of passes is not known at the outset. Instead it will run as long as some logical condition is true. In a common type of convergence loop a set of calculations are repeated ending with a test or error calculation being made. The program continues to

make passes through the loop with each pass having a value get closer to the correct result. The difference between the calculated value and the correct value would be the error. The repetition block would stop when this error drops below some acceptable value. It is unlikely that you would know the number of passes that the loop will need to make before the acceptable error is achieved so instead it runs until the calculated value converges to the acceptable value. Thus a convergence loop.

Convergence Loop
The more common form of repetition is the *convergence loop*. It is a type of repetition structure in which a block of code is repeated until some condition is met. The number of passes that the loop makes might be determined prior to the start of the loop - such as in a counting loop - but it is also common that it is not known until the loop is complete.

A non-computing example of a convergence loop is a little leaguer trying to throw a baseball into a trash barrel ten meters away. He will continue throwing the ball towards the barrel until he gets one in. Once he does, he stops. He might get it on the first throw or the one hundredth. No one knows how many throws he will need to make - or in the programming vernacular - how many passes the loop will require until he actually makes it.

Another example of convergence is simple counting. It is not uncommon that we need to repeat a block of code five time, or ten times, or perhaps even ten million times. The number of passes through the block of code is not relevant. What is important is that the number of passes through the loop is know before the loop starts. It is still convergence because the loop repeats until some counter value is met - in effect until the counter converges to the predetermined value.

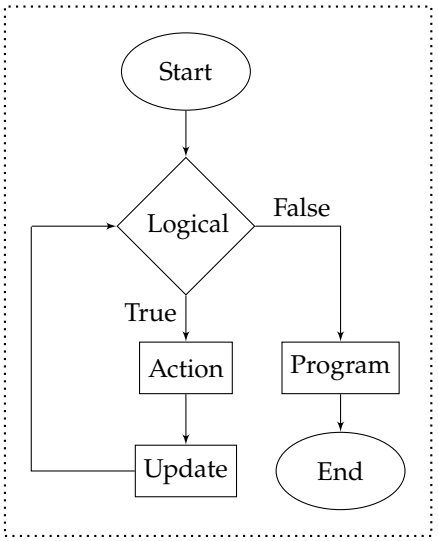
An example of this type of repetition is providing directions. A driver pulls over and asks you for directions. You point them in the correct direction and tell them to go through five traffic lights and then turn right at the sixth. The driver leaves and when they get to the first light they count one and continue driver (another pass through the loop). They repeat this for light two, three, four, and so on. If you were asked for the directions by other drivers you would always tell them the same number of traffic lights. In this second example the convergence is that you repeat until you have completed some task a fixed number of times.

In both of the examples, the loop will need to make some type of test to determine if the goal has been met - or at least is it close enough? A question that will arise is *should the test be made before the first pass through the loop , or after the loop has run the first time?* Which is the difference between a *pre-test loop* and a *post-test loop.*

9.2.1 Pre-Test Loop

Pre-Test Loop
A pre-test loop is a repetition structure in which a logical decision is made before the loop begins. If it is true then the block of code is run. But if it is false the block of code is skipped and the program continues not having run the loop at all.

The pre-test loop is more colloquially known as the *while* loop. It operates by first checking a logical condition. If the logical condition is **true** - or 1 - then it makes a pass through the block of code that follows it. Once the pass is complete it returns to the logical test and the process repeats. In effect, the loop will continue *while* the logical condition is true - thus the name *while loop*. Eventually the logical condition will have to be **false** - or 0 - when this happens the program skips the block and continues on with the program. The syntax of the *while loop* is very similar to that of the simple *if* selection structure. This syntax is shown in figure 9.1.




```
1  while(logical)
2    % Code to run if the
3    % logical is true
4
5    % Update
6
7  end
```

Figure 9.1: Flow Chart and MatLab Syntax of a Pre-Test Loop

An important component of the *while* loop is the update. Each time that the program makes a pass through the block of code that forms the loop there must be the possibility that the logical condition becomes **false**. If it did not then the loop would never stop and we would have the run-time error known as an *infinite loop*.

while loop
while(logical) {
%Code if true

%Update

end

To ensure that the loop does eventually end there must always be an update within the block of code. This can be an increment to a counter, or a check on error term, or any of many other means of having the logical condition change state from **true** to **false**. The method of the update is not as important as the understanding that it must exist within the *while* loop's block of code.

Although the *while* loop can be used for either a convergence process or an iterative one. it is the ideal structure when we want the process to converge. The introductory example of throwing a baseball was an example of repetition until convergence is achieved, but we are not writing a program about throwing a baseball - although we could.

Example
Calculating a square root.

A technique that can be used to estimate a square root involves the Newton-Raphson Methos. In it, a starting value is guessed for the square root of s. We can use the value $x = 1$. If $x^2 - s \neq 0$ then x it is not the square root and we calculate a new estimate using 9.2.

$$x_1 = x - \frac{x^2 - s}{2x} \tag{9.2}$$

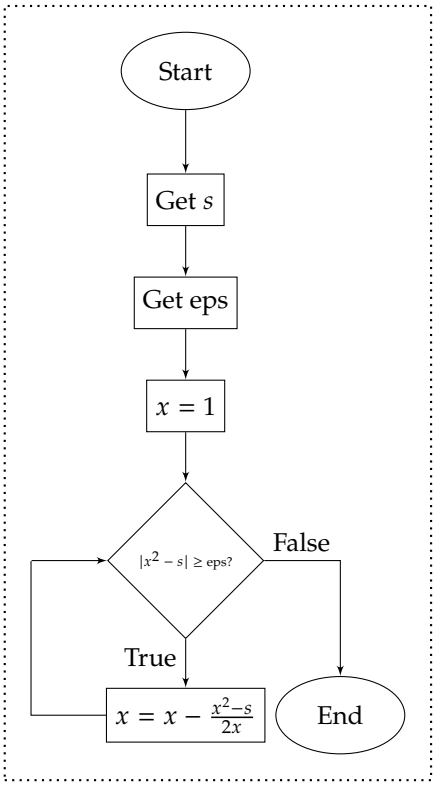
The convergence criteria is that $x^2 - s$ be zero which would mean that $x = \sqrt{s}$. So how do we code this using a convergence series?

Take note that when the function is implemented it first checkis if the starting value is in fact the square root - or at least within some small distance from it. If it is, for example if the value of the square that is passed to the function were $s = 1$ then the value $x = 1$ would be the square root and the loop would never run - or need to since it already has the square root.

The while loop in the example demonstrates the *pre-test loop*. Before any steps in the loop are run a test is made. In this case the test is if the starting value when squared is close to the value that was passed to the function. If the test is true - the value is far from the squared value - the code in the loop block is run. But if it were close then the test would have been false and the block would have never run.

Assuming that the test result is true the block runs, performs its calculations, may update some test value, and then returns to the logical test. This process will continue until the test eventually returns false and the algorithm ends. This characteristic of the pre-test loop can be described as *test - run - repeat*.

The test is the beginning creates an important consideration in the pre-test loop. Since the logical test is always done first, if it is false on the first test the loop would never run. That the




```
function [x] = square_root(s, eps)
%    SQUARE_ROOT [s] = square_root(x, eps)
%    estimates the square root of the
%    value x  using the Newton-Raphson
%    method

     % Error check the inputs
     if((s < 0) | (eps <= 0))
          error('Error on input: out of
    range');
     end

     % Set a starting value (a guess) for
     % the square root
     x = 1.0;

     % Enter loop - make passes till it
     % converges
     while(abs(x.^2 - s) > eps)
          % Calculate next term
          % This is also the update term
          x = x - (x.^2 - s) ./ (2.*x);

     end
end
```

```
>> [s] = square_root(2, 0.001)
s =
    1.4142
```

Figure 9.2: Flow Chart and Code Sample of a Function to Estimate the Square Root of a Value

loop might never run is an important design consideration in using a pre-test loop.

An alternative to this is to design a repetition structure that will always run at least one time. This type of repetition structure is known as a *Post-Test Loop*.

9.2.2 Post-Test Loop

Post-Test Loop
A post-test loop is a repetition structure in which a logical decision is made after the pass through the loop. If it is true then the block of code is run again. If it is false the structure ends and the program continues. The post-test loop will always run at least one time.

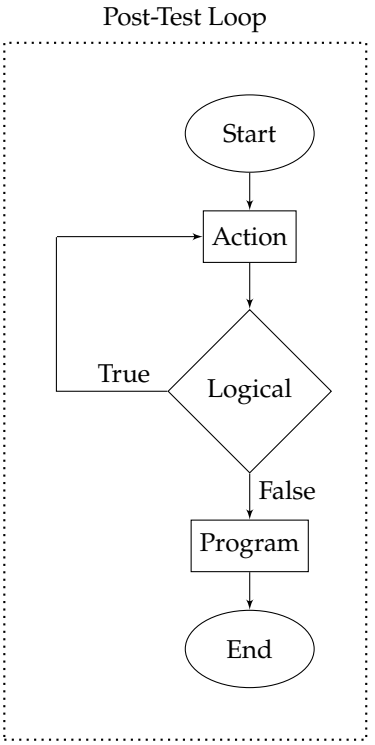
The *pre-test* loops is just what is says - the loop performs a test before the loop begins. The result of the test determines if the program should make a pass through the loop or not. If the

test is true - the pass is made. If it is false the loop block is skipped and the program continues. So how is this different from a *post-test loop?*

The sole difference is that the post-test loop makes a pass through the block of code and then it performs the logical test to determine if it should do it again. The main functional difference is that while a pre-test loop may not run at all, a post-test loop will always make at least one pass through the block of code that forms the loop.

The post-test loop is shown schematically in the flow chart of the post-test loop in figure 9.3, but MatLab does not have a post-test structure. So how would you implement this? You would adapt a while loop to function as a post-test loop. This does not imply that there is a post-test structure, but just that it can be mimiced by adapting something else.

You when you look at the syntax it shows another while loop - a pre-test loop. This is because MatLab does not implement a direct post-test loop. Thus you would have to adapt the while loop.




```
% Statement to force logical
% to be true
test = 1;
while(test)

  % Code to run if the
  % logical is true

  % Update
  % The update will need a
  % test = 0; at some point
  % in the loop

end
```

Figure 9.3: Flow Chart of a Post-Test Loop with and adaptation of a while loop to mimic the Post-Test Loop

Since the only convergence loop available in MatLab is the while loop, you will need to adapt it to act as a post-test loop. This requires that you force the logical test to be true the first time that it is encountered. A common way of doing this is to set an initial value for the variable that will be updated in the block of the loop. As an example, if the logical test is that the loop should run as long as the value stored in the variable **x** is greater than 10, then you set an initial value of $x = 11$, or some value that is greater than 10. Doing this will ensure that the loop always runs the first time that it is encountered.

Example

Earlier, we had developed an error checking function by using recursion. An alternative method is to use a post-test loop. The reason for the post-test loop is that the block of the loop will contain the input function for the user to enter data. As a result the block of the loop must always run at least once or there would be no way for the user to enter data.

Note

We could have made the initial value positive infinity just as easily as we did negative infinity, but it is more common for an input to have a lower bound while being unbounded above than it is to be unbounded below while having no upper bound, so this will usually be the better choice. An alternative is to use a flag that is initially set to true, then if the entered value is within the range the flag would be set to false.

Because we do not have an actual post-test loop, this function will still require the use of a while loop - a pre-test loop. We adapt this by adding an initial value that will always result in the first test of the logical condition returning true. This is a common way to force a pre-test loop to act as a post-test loop.

The flow chart, figure 9.4, shows this initial condition to be negative infinity - a clearly small and out of range value.

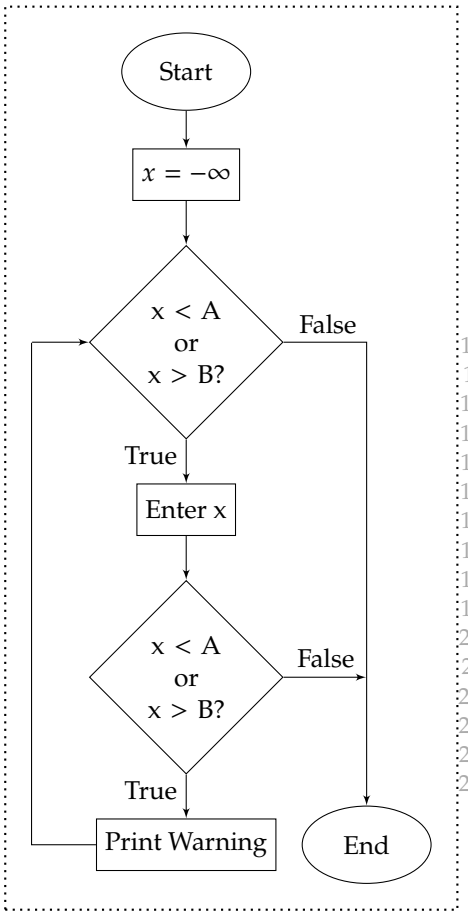
When the user calls this function they would pass three variables; a string that will be the prompt, the lower bound, and the upper bound. The first pass through the loop - which will always happen - prompts the user to enter a value. It then uses a simple if selection structure to test if the value entered is out of the range. If it is the warning function is called printing a message to the user, and the pass ends and the flow returns to the test logical test. Since the value is already out of the designated range, the logical test simply repeats this and it will of course be true and another pass will be made through the loop.

When the user does enter an acceptable value - whether it is the first time or the fifteenth - the simple if test will be false and the warning will be skipped. The logical test within the while statement will also be false and the loop will end.

Note

We had earlier written a function in which we check the entered value to be sure that it is within a set range. At that time, in chapter 8, we used recursion. The recursive approach does not require the two sets of tests each time, or setting the initial value to force the loop. You could argue that the simplicity of the recursive function makes it a preferred choice.

Convergence loops can function either pre-test or post-test. Regardless they follow the same algorithm - passes are made through a block of code until something result forces the loop to end. This could be a value for a counter or a sum is




```
function [x] = get_data_range(prompt, m, M
    )
% GET_DATA_RANGE [X]=get_data_range(prompt
    ,
% m, M)
% has the user enter a floating point
    value,
% then error checks that it is within the
% range of m to M. If it is not it prints
    a
% warning and starts again.
%
% This is a forced post-test loop

 % Set x to force the loop to run
 x = -inf;

 % Enter loop - will be true on first test
 while((x < m) || x > M))
 % enter x using the prompt
 x = input(prompt);

  % test if out of range
  if((x < m) || x > M))
   % print warning
   warning('Value out of range');
  end
 end
end
```

```
>> x = get_data_range('Enter value from 0
    to 10: ', 0, 10)
Enter value from 0 to 10: 50
Value out of range
Enter value from 0 to 10:
```

Figure 9.4: Flow Chart and Code Sample of a Function for entering data and then error checking it to ensure that it is within a pre-determined range

reached, or a mathematical operation gets within an acceptable distance from a value. But there is another possible type of repetition in which the program makes a pass through a block of code one time for each item in a list. This is known as *iteration* and the loop is an *iterative loop*.

9.3 Iterative Loop

There are many examples where a task is repeated once for each item in a set or list. In this case the the number of times that the pass through the block of code is known before the loop begins. while similar to a counting loop, this is different in that there are a specific set of items or elements in a list that well be accessed with each pass. This type of repetition is known as *iteration*.

Iterative Loop
An iterative loop is a repetition structure in which a set or list of items is first created. The iterator is then set to each item in sequence making a pass through the loop for each one.

9.3.1 For Loop

For Loops

In MatLab the iterative loop is known as the *for* loop. It operates by creating a list of elements. It then steps or *iterates* though each item in the list. With each pass through the loop a variable - known as the iterator - is assigned the current element in the list. After the iterator has been assigned each element the for loop exits. The syntax of the for loop is shown in figure 9.5.

The iteration is done automatically by the interpreter. So while the flowchart in figure 9.5 shows a step for having the iterator move to the next element you do not actually program this. Instead the *for* loop is structured as only the single line of code at the beginning of the block of code.

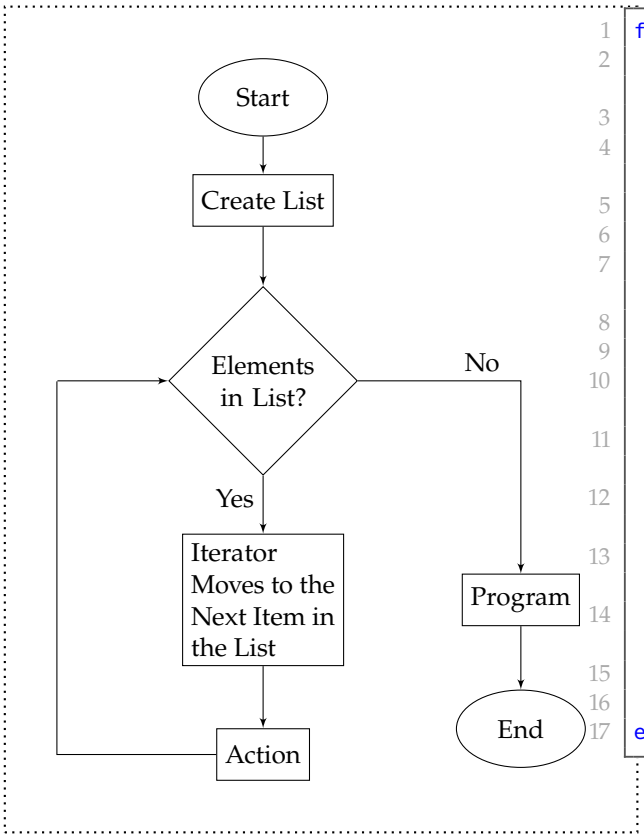
The MatLab implementation of the *for* is reasonably basic. It has only two components - the iterator and the list.

Iterator

In a *for* loop, the iterator is a variable that points to a single element in a set or list of elements. When the for loop is first created the iterator is assigned the first element in the list. At the completion of each pass through the *for* loop the iterator moves - or points - to the next element. This continues until the iterator has pointed to each element. It is because of its role in pointing to the current element of the list that the iterator in a for loop is also called a *pointer*.

Iterator
An iterator is a pointer that maintains the value and location of an element in a list or set of elements.

For our purposes the iterator in the *for* loop looks like a variable and acts like a variable, but is actually much more. While it retains the value of the current element in the list - a variable - it also maintains the location in the list - a pointer.




```
for k = [list]
  % In the for loop, k is an
  % an iterator. It is
  % assigned a single value
  % from the  list one
  % element at a time
  % in the order in which
  % they are presented.

  % The action is the code
  % that is run in the block
  % of the loop. It will
  % continue to run as long
  % as there are items still
  % in the list

end
```

Figure 9.5: Flow Chart and MatLab Syntax of a For Loop

While this is important, for us we still use it as if it were a variable.

List

The second component in the *for* loop is the list of elements. This can be called by several different names, *list*, *array*, *vector*, or simply *set*. In MatLab is often compared directly to a vector, but much the difference between an *iterator* and a *variable*, there are differences between a *list* in a for loop and a *vector*.

List
A list is a set of elements. It is also known as a *vector*, an *array*, or a *set*.

Element
An *element* is a single item in a list.

The list is the set of elements that will be assigned as a value of the iterator each time that it makes a pass through the loop. The list can be created in several different ways, but in its most common form the list is created by enumeration inside

a set of square brackets, [and].

If you have a specific set of values of which you would like to assign to each pass through the loop then you create a list consisting of the elements. When part of a for loop, the list and the iterator take the form

```
k = [e_1, e_2, e_3, e_4, ... , e_5]
```

Figure 9.6: Creating a list using enumeration

where e_k is some numerical value or a character. The elements may be comma delimited or simply space delimited. Note that when this is used in a for loop, while it appears to be similar to an assignment statement it is not, Thus it should not have a semicolon at the end.

An example of using enumeration to create the list of elements in a for loop is in figure 9.7 .

```
fprintf('k = ');
for k = [6, 2, 7, 8, 5, 2]
    % Statements to be run with each pass
    fprintf('%d ', k);
end
```

```
k = 6 2 7 8 5 2
```

Figure 9.7: Example of a for loop using enumeration

This approach to creating the list is known as *enumeration*. It is a useful technique when the number of elements is small or the elements that need to be accessed are not ins numerical order.

Enumeration
Enumeration is the process creating an ordered collection of all the items in a list.

The elements enumerated in a list are not limited to numerical values. They can also be characters.

```
fprintf('k = ');
for k = ['c', 'r', 'w', 'a', 'n', 's']
    % Statements to be run with each pass
    fprintf('%c ', k);
end
```

```
k = c r w a n s
```

Figure 9.8: Example of a for loop a list of characters

An interesting result will occur if the elements are strings of text. In MatLab a string is already a list of characters. This

results in the elements being individual characters instead of the strings.

In the sample code, each character of the two strings is interpreted as an element. This for loop would thus make ten passes through the loop instead of two.

```
fprintf('k = ');
for k = ['Hello', 'World']
    % Statements to be run with each pass
    fprintf('%c ', k);
end
```

```
k = H e l l o W o r l d

```

Figure 9.9: Example of a for loop with strings

Enumeration does have a downfall - size. Hardcoding a handful of elements in to a list is not a major undertaking. But for loop with hundreds, or thousands, or even millions of elements is not uncommon.

If the elements are ordered, then an alternative to enumeration is to create the list of elements as a range of values. The list will begin at a fixed starting value, and increase up (or decrease down) to a stopping value with each additional element increasing (or decreasing) by a fixed amount - the step size - from the previous value.

```
k = [start:step:stop]
```

Figure 9.10: Creating a list using a range of values

There are several conditions that are applied when creating a list by range;

1. The step parameter is optional. If it is omitted it defaults to 1

```
fprintf('k = ');
for k = [5:12]
    fprintf('%d ', k);
end

```

```
k = 5 6 7 8 9 10 11 12

```

Figure 9.11: List example with step = 1

This creates a problem if the stop value is smaller than start. Since the default value for step is one, the list would consist

of all of the elements from *start* decreasing to *stop* but with the difference between them being a positive number. This is impossible so the list will be empty. It will still be created, and the program will run, but since the list is empty there are no values for the iterator. No elements means that the loop will never run.

If the goal was to not create an empty list, you must explicitly set the step size. In this example the step is set to −1 so that the iterator will decrease by one with each pass.

```
fprintf('k = ');
for k = [9:-1:3]
    fprintf('%d ', k);
end

```

```
k = 9 8 7 6 5 4 3

```

Figure 9.12: List example with step = -1

2. The step size will determine the distance between each element. If the step is positive then the elements increase while if the step is negative the elements decrease.

```
fprintf('k = ');
for k = [2:5:27]
    fprintf('%d ', k);
end

```

```
k = 2 7 12 17 22 27

```

Figure 9.13: List example showing the effect of the step value

If the step contradicts the order of start and stop, ie [8:2:-3], the list will be created but it will be empty. As with the missing step size, a for loop using this list will be created and the program will run, but the loop itself will not make any passes.

3. The list will always terminate at the value at or below stop. Thus if the the range of values beginning at start and increasing by step does not include the exact value of stop then the terminal value will below the selected stop value.

```
fprintf('k = ');
for k = [18:3:0]
    fprintf('%d ', k);
end

```

```
k =

```

Figure 9.14: List example showing the list will not pass the stop value

```
fprintf('k = ');
for k = [2:3:12]
    fprintf('%d ', k);
end

```

```
k = 2 5 8 11

```

Figure 9.15: List example where the list will stop short of the stop value

There is no requirement that the start, step, or stop values be integers. They can be assigned any value.

```
fprintf('k = ');
for k = [0.9:3.7:8.5]
    fprintf('%d ', k);
end

```

```
k = 0.9 4.6 8.3

```

Figure 9.16: List example with non-integer values

Since the loop is designed to iterate through a fixed set of values the Gaussian sum is best implemented by using a *for* loop. But this is not the only implementation. We could also have written this code using a *while* loop.

It is possible to replace every *for* loop with a *while*. The challenge would be to replace the list of elements with a logical test and an update. This could be done by first determining the number of passes that the loop would need to make.

Calculating Number of Passes and Step Size

If you need to know how many passes a for loop makes, you can add a counter that updates within the loop. But it is

important to be able to know how many passes a for loop will make before it is run. If the elements in the list were enumerated then you already know. But if you are using the range approach then number of elements is not so obvious.

The number of steps that is made is simply the integer value of the distance from the lowest value to the largest value divided by the step size. But there is one more element than there are steps - the first and last element *bookend* the list. Thus

$$n = \text{floor}\left(\frac{\text{stop} - \text{start}}{\text{step}}\right) + 1 \tag{9.3}$$

This equation will return the number of elements of a list where the start, stop, and step are known. It often occurs that you want the list to contain n elements equally spaced from start to stop. The equation in 9.3 can thus be solved for the step size.

$$\text{step} = \frac{\text{stop} - \text{start}}{n - 1} \tag{9.4}$$

Using equation 9.4 you can create a for loop with n elements equally spaced from start to stop by

Figure 9.17: Creating a list a fixed number of elements

```
k = [start:(stop - start)./(n-1):stop]
```

The approach in 9.17 is common enough that MatLab provides a function that does the heavy lifting for us. It is the **linspace** function.

Figure 9.18: Creating a list a fixed number of elements

```
k = linspace(start, stop, n)
```

9.4 Accumulators

linspace
linspace creates a list of n elements equally spaced from start to stop
syntax: **k = linspace(start, stop, n)**

Repetition is the process is repeating a set of operations until some criteria is met - either a value converges, or perhaps a flag becomes false, or a list of elements is exhausted. The operations themselves are as varied as the applications of the loop. There is, however, an application of repetition that is so

common that it warrants its own presentation; maintaining a running sum or total. In this application a variable is initialized before the loop, and then with each pass the results of a calculation are added - or subtracted, or multiplied, or divided - from the variable.

This variable is known as an *accumulator*. It is a variable that is initialized before the onset of a loop. Once the loop begins the value in the accumulator will be updated by the results of some operation that took place in the loop. In its more common use the accumulator is a counter. It keeps a running count of the number of passes that have been made through the loop.

Accumulator
An accumulator is a variable that is initialized before the implementation of a repetition structure and is then updated with each pass through the loop.

But counters are not the only application of accumulators. They can be used for any application in which a running total, or sum, or product, needs to be maintained. While a single accumulator is often included with a loop, it is not uncommon for a loop to have multiple accumulators. This is often the case when you need to keep not just the count of the number of passes through the loop but sum of the operations.

An application of an accumulator as a sum is the calculation of the Taylor Series expansion of a function.

Example

Recall from the Calculus that the Taylor Series is an infinite series that converges to some value of a function on a particular interval. A common example is the geometric series, equation 9.5.

$$\begin{aligned} f(x) &= \frac{1}{1-x} \\ &= 1 + x + x^2 + x^3 + \cdots \\ &= \sum_{k=1}^{\infty} x^k \end{aligned} \tag{9.5}$$

This series converges as long as $|x| < 1$.

The series itself is infinite, but a sum of a finite number of terms of the infinite series will approximate the Taylor expansion. In writing code to do this approximation, there will be three items to address: How we do keep track of the current power of the term? How do we store the sum? And how many passes through a loop will be needed?

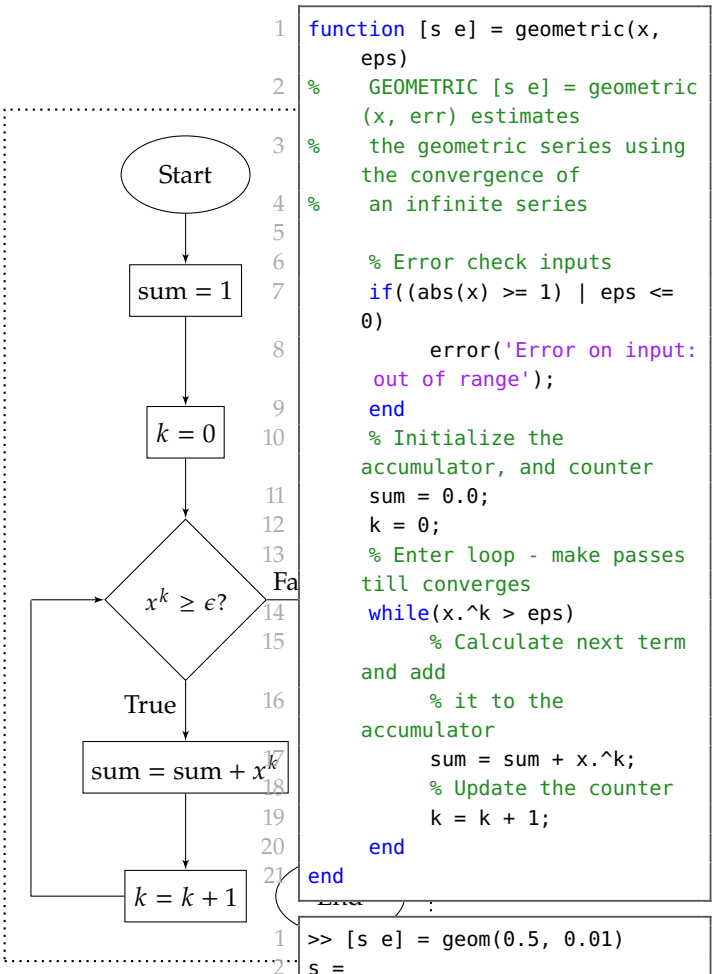
For the first we will create an accumulator to be the counter. This will be created outside of the loop and initialized to 0.

For the second item, since it is a finite sum, we can create another accumulator to store the running sum. But this accumulator will be initialized to 0 before the first run of the loop. This is so that the first term of the series - which will always be 1 - when calculated outside of the loop will be compared to epsilon and will most likely be true so that the loop will run.

As to the number of passes - since the sequence is decreasing (remember that $|x| < 1$ thus a fraction so each term is smaller than the one before it) we should continue until the amount that the sum increase by the additional term is below our cutoff. This just means to continue until the next term to be added is below some value that is passed to the function by the user - an epsilon.

We could use a for loop if the user knows how many passes will need to be made before the loop starts. In this case we would create a list $k = [0 : n]$. But that would be inefficient. Why make more passes than are needed. Instead, since we do not know the number of passes that will need to be made, we should use a convergence loop. This is shown in figure 9.19

The Taylor expansion of the geometric function can be adapted to any continuous function. All that it takes is to change the terms of the sequence.




```
function [s e] = geometric(x,
    eps)
%    GEOMETRIC [s e] = geometric
    (x, err) estimates
%    the geometric series using
    the convergence of
%    an infinite series

     % Error check inputs
     if((abs(x) >= 1) | eps <=
    0)
          error('Error on input:
     out of range');
     end
     % Initialize the
    accumulator, and counter
     sum = 0.0;
     k = 0;
     % Enter loop - make passes
    till converges
     while(x.^k > eps)
          % Calculate next term
    and add
          % it to the
    accumulator
          sum = sum + x.^k;
          % Update the counter
          k = k + 1;
     end
end
```

```
>> [s e] = geom(0.5, 0.01)
s =
    1.9922
e =
    0.0078
```

Figure 9.19: Flow Chart and Code Sample of a Function to Calculate the Taylor Expansion of the Geometric Series

9.5 Nested Loops

The convergence and iterative loops developed have all been single loops. This means that there is only a single repetition structure and a single set of passes. This works if, for example, a counter, x, needs to go from one to n. After making n

passes it finishes and continues on with the program.

Imagine a second example - analyzing each point on a plane. This would require performing calculations over a two dimensional space - each (x, y) coordinate.

To do this you would have the program would start a corner - say $(0, 0)$. The step through each x value until it reaches its maximum. At this point you have the program move y from 0 to 1 and then reset x back to 0. From here the process repeats. This continues until the value of y reaches it maximum value.

Each value of x will require a loop and also each value of y. But the x loop must run *for each* value of y. This will require that the x loop be embedded - or *nested* inside of the y loop.

Nested Loops
A loop is nested when it is contained completely inside of another loop.

A flow chart to show the process of nested loops is shown in figure 9.20. When the repetition structure is entered, any needed initializations for the outer loop will be made. A logical test for the outer loop is now made. If it is true then the inner loop is entered.

The inner loop is almost the same as the outer loop. Any initializations for the inner loop variables are made, and then the inner loop logical test is done. If it is true then a pass through the inner loop is made. In the inner pass, any updates that will eventually effect the inner logical test will be made.

There are no restrictions on the types of loops that can be nested. A while loop can be nested within another while loop. A for loop can be nested within a while loop or vice versa. For loops can be nested within for loops. The for loop inside of a for loop is a common implementation of nested loops for working over a Cartesian plane - it is the example that introduced nested loops in this section.

An example of a function that implements a for loop nested within another for loop is shown in figure 9.21. In this example the function steps through each possible integer value of x, then updates y, and repeats stepping through x again. This continues until every (x, y) pair on the plane has been accessed.

There is no fixed limit to how deep the nesting of repetition structures can be. Two loops - an inner loop nested within an outer loop is common. As are having three loops nested one inside of a second inside of the outer, or third, loop. This example is one in which you might be accessing each coordinate point in a three dimensional space.

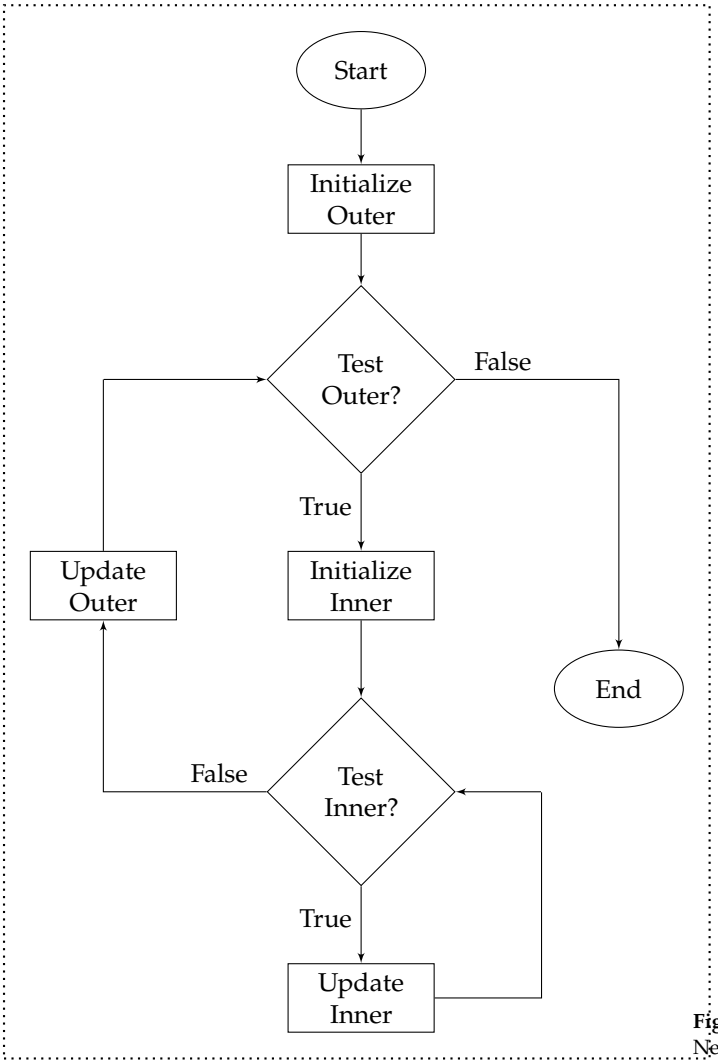


Figure 9.20: Flow Chart of a Nested Loop

```
function cart_plane(x_min, y_min, x_max, y_max, x_step, y_step)
   % Check nargin for number of variables
   if(nargin == 0)
      % Set start, stop, and step to default values
      x_min = 0;
      y_min = 0;
      x_max = 100;
      y_max = 100;
      x_step = 1.0;
      y_step = 1.0;
   elseif(nargin == 4)
      % Start and stop are passed to the function
      % Set step to default of 1.0
       x_step = 1.0;
       y_step = 1.0;
   else
        error('Incorrect number of input parameters to function cart_plane'
    );
   end

   % Start the outer loop
   for y = [y_min:y_step:y_max]
      % Any operations for outer loop only
      % This is commonly empty but not always
      for x = [x_min:x_step:x_max]
         % All operations to be done the the point (x, y)
      end
   end
end
```

Figure 9.21: Sample Function of Nested For Loops

9.6 Efficiency and Complexity in Repetition Structures

Nesting loops can be an effective means of performing multiple calculations without have to rewrite code. Simply have the same operations performed with each pass through the loop. The issue - if there is one - is that of time.

If a program makes n passes through a loop, where n is the number of pieces of data, then as we have seen before this algorithm would be $O(n)$, that is it would have linear complexity.

If you nest two loops and the inner makes n passes while the outer loop makes m, then there would be a total of $m \cdot n$ passes made. This would be a quadratic time algorithm, $O(mn)$ or more commonly $O(n^2)$.

Continuing with this, three nested loops would be expected to be $O(n^3)$, while four is $O(n^4)$. If we extend this to p loops nested one inside of another it is $O(n^p)$. All of these are still polynomial time algorithms and should not be of great concern. But if the number of passes is large - and millions is not unusual - the time requirements could be noticeable and perhaps severe.

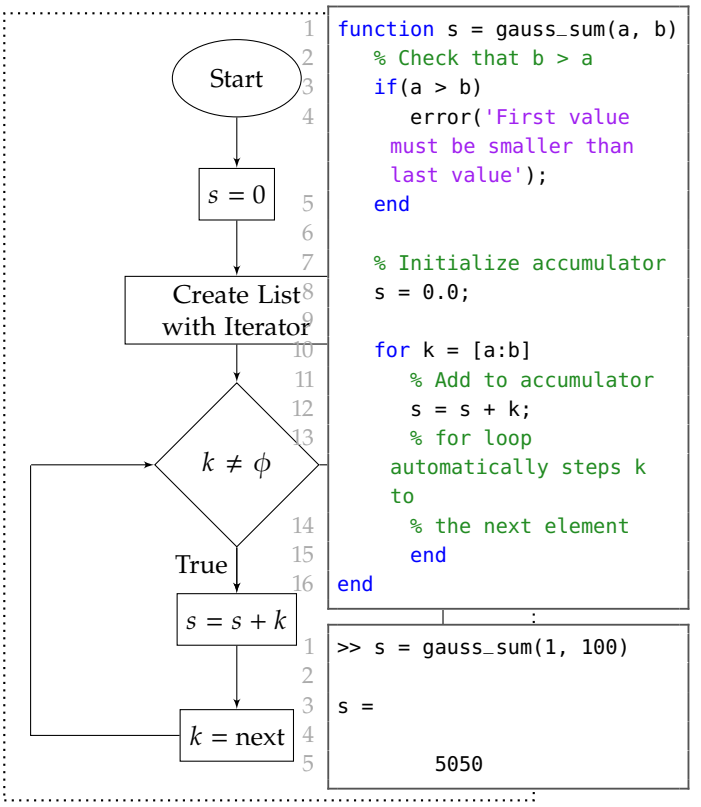
As such, while repetition is a common technique in programming, if it is possible to eliminate a loop then it should be done. An example of this is the Gaussian sum that we used to introduce this chapter.

Example

The Gaussian sum is an application that uses a single pass repetition structure. Because it involves adding all of the integers from a starting value a to a maximum value b, it is a common implemented using a *for* loop.

To calculate the Gaussian sum we need to create an accumulator for the sum and set it to zero. The function will then have to create a list of the elements from 1 to 100. The *for* loop will take care of iterating through the list. Thus the only requirement for each pass through the loop is to update the accumulator with the next value in the list.

While this function will run quickly for almost any minimum and maximum input values - it is a $O(n)$ or linear time




```
function s = gauss_sum(a, b)
   % Check that b > a
   if(a > b)
      error('First value
     must be smaller than
     last value');
   end

   % Initialize accumulator
   s = 0.0;

   for k = [a:b]
      % Add to accumulator
      s = s + k;
      % for loop
     automatically steps k
     to
      % the next element
      end
end
```

```
>> s = gauss_sum(1, 100)

s =

        5050
```

Figure 9.22: Code Sample of a Calculating a Finite Sum Using a For Loop

algorithm - it can be reduced to $O(1)$ - a constant time algorithm - as Gauss did when he was nine years old.

Gauss observed that if instead of adding each element of the list individually to a running total - his accumulator - you could simply add the first and last values, in this case $a + b$. The same sum occurs if you move one up with a and one down with b, $(a + 1) + (b - 1) = a + b$. This continues for every pair of values of which there are $\frac{b-a+1}{2}$ pairs. Using his observation Gauss determined that

$$s = \frac{(b - a + 1)(a + b)}{2} \tag{9.6}$$

This new equation can be written as a function without any repetition.

```
function s = gauss_sum(a, b)
   % Check that b > a
   if(a > b)
      error('First value must be
     smaller than last value');
   end

   % No accumulator

   %    No Loop - just a single
     equation
   s = (a + b).*(b - a + 1)./2;

end
```

```
>> s = gauss_sum(1,
     100)

s =

        5050
```

Figure 9.23: Flow Chart and Code Sample of Calculating a Finite Sum Without a Loop

Although a repetition structure is not normally a problem in a program - they do create the ability to perform the same calculation hundreds or even millions of time - they can create time issues.

A single loop has linear complexity, while two loops nested one within the other would be $O(n^2)$. As the number of passes increases these loops can start to slow down the program. While it is not common that a loop can be replaced with a single calculation, if it can - as the Gauss Sum example shows - it can greatly improve the efficiency of the program.

9.7 Errors in Repetition Structures

There is a saying in real estate that the majority of the complaints that home owners have about home ownership can be put into two groups; *no water where you want it*, and *too much water where you don't*. We will add a third - *the water pressure is just a bit too low, or perhaps just a bit too high.* These three complaints can be adapted almost directly to repetition and loops

How are water and repetition structures related? No water where you need it is analogous to a loop that does not run. Too much water where you do not want it is similar to a loop that never stops. And too low - or too high - water pressure relates to the loop running but stops too soon - or runs just a bit too long. These are commonly described as *a loop that never runs*, an *infinite loop*, and the *off by one error*.

9.7.1 Loop that never runs

The first type of possible error in implementing repetition is that the loop never runs. The best designed repetition algorithm is useless if the program never makes a pass through the loop. Instead of making passes through the loop, the logical test fails and the block of code in the loop never runs. Since the program still runs - and runs to completion - it would most likely result in an incorrect result. As such this would be a *logic error*.

A problem with the loop is to determine whether or not it actually runs. With modern processors program can run so fast that the user will not notice the difference between the program making a few hundred passes through a loop and not running the loop at all. So how do we know that the program is not making passes through the loop?

Identifying the error

Trace
A trace is a temporary programming line used for debugging - often a print statement - that indicates what is happening at a specific part of the program.

An easy way to determine if the loop is running is to add a *trace* to the block of code in the loop as is shown in figure 9.24. Recall from Chapter **??** that a trace is just a line of code, usually a print statement, that provides information to the programmer about the code at that point. It is meant as a temporary addition to the program that will be removed - or switched off - when the program is complete.

Adding a trace that prints *Made it here* to a loop to determine if the loop is running is a simple way to determine if passes are being made. If the loop is running then *Made it here* will print each time that a pass is made through the loop. If it does not print at all then the loop is not running.

Causes of the error

There are two actions that occur before a pass through a loop that will directly impact whether or not the loop runs; initialization of variables and, for a convergence loop, the logical test, while for the iterative loop the instantiation of the list of elements.

The initialization of variables is a possible cause of the loop never running. If you design a loop to run as long as $x < 10$, you would need to initialize the variable x to some value less than 10. But imagine that while you wanted $x = 0$ to start, you erroneously set $x = 10$ - a common error is switching the initialization value and the test condition. Or using the

```
function check_loop(x, a)
%   CHECK_LOOP check_loop(x, a)
    is designed
%   to run as long as x < a

   %   Set global debug
   global debug = 1;

   %   Enter loop
   while (x < a)

      %   Use the debug method
    for the trace
      if(debug)
         fprintf('Made it here\n'
    );
      end
      %   Update x
      x = x + 1.0;

   end
end
```

```
>> check_loop(5, 8)

   Made it here
   Made it here
   Made it here
```

Figure 9.24: Code Sample of Using a Trace to Determine if a Loop is Running

correct initial value but setting the test condition to $x < 0$. In either case the initial logical test would fail and the loop would never run.

9.7.2 Infinite loop

The second error is that the loop never stops. This is so common that it has has its own name, the *infinite loop*. When it starts the loop will continue to run until the user intervenes. In MatLab this is done by using *Ctrl C* to stop the program.

Infinite Loop
An infinite loop occurs when a repetition structure begins but never stops.

Since the program runs but never returns an result, an infinite loop would be considered a *runtime error*.

Identifying the error

Ending an infinite loop
The user can stop a program that is in an infinite loop by pressing Ctrl C.

Identifying an infinite loop can be as simple as recognizing that the program is running for an unusual amount of time. This can be problematic because a loop that has to make many passes may be mistaken for an infinite loop.

We can use the trace from before to identify the infinite loop. If we do the *Made it here* will continue printing on the screen. An improvement to this is to make the two changes in

```
function check_loop(x, a)
%   CHECK_LOOP check_loop(x, a)
    is designed
%   to run as long as x < a

   %   Set global debug
   global debug = 1;
   %   Create a counter
   counter = 0;

   %   Enter the loop
   while (x < a)
     %   Update counter
     counter = counter + 1;
     %   Use the debug method for
     the trace
      if(debug)
         fprintf('Pass Number %d\
    n', counter);
      end
      %   Update x
      x = x + 1.0;

   end

   %   Print a completed step
   if(debug)
      fprintf('Completed %d
    Passes\n', counter);
   end
end
```

```
>> check_loop(5, 8)

   Pass Number 1
   Pass Number 2
   Pass Number 3
   Completed 3
    Passes
```

Figure 9.25: Code Sample of Using a Trace to Identify an Infinite Loop

figure 9.25. The first is to add a counter to the trace so that you know the number of passes made through the loop. The second is to add an additional trace statement after the loop. This second print statement shows the total number of passes through the loop

Causes of the error

A common issue that can cause an infinite loop is the update - or more specifically the lack of an update. If the update step is missing from the loop block then the logical test will never have a chance to become false. Thus the loop will never end.

A second still involves the update step. In this case the update takes the test value in the wrong direction. For example, if the test is that $x < 10$ with an initialization of

$x = 0$, and an update $x = x - 1$ then then value will move farther away from ending instead of closer. In this case the correction is that the update should be $x = x + 1$.

9.7.3 Off by one error

There is a third possible error in working with loops - one that could be the most problematic of the three. It is commonly known as the *off by one error* but in general it means that the loop runs and stops, but it did not run the correct number of times. While the name implies that when the loop runs there is one too few or one too many passes made, in actuality it could be any incorrect number of passes. Similar to the loop that never runs, the off by one error would be considered a *logic error*.

Off By One Error
An off by one error occurs when a repetition structure too few or too many times.

Identifying the error

Identifying the off by one error is done in the same way as identifying the infinite loop. By using the same debugging traces in figure 9.25 you will get a count of the number of passes that were made through the loop. If the loop should have made ten passes, but only made nine then the counter will show nine.

Cause of the error

The off by one error is commonly a result of an incorrect logical test. If the loop makes one too few passes it can often be corrected by changing an absolute inequality to an inequality. This means that if the logical test is $x < a$ then it might need to be $x <= a$. On the opposite side if the loop makes one too many passes then a logical test $x <= a$ should probably be $x < a$.

A second, although less common, problem is the update step. This might be the cause of the off by one error if the loop is making half - or a third, or a quarter - the number of passes than it should. Since a common update is $x = x + a$ the problem might the value a. Instead the update might need to be $x = x + 2. * a$ or some other multiple.

9.7.4 Errors with an Iterative Loop

Two of the three repetition errors are possible with the iterative loop (the for loop). These are that the loop never

runs, and the off by one error. Since it is impossible to create an infinite list you cannot have an infinite loop with a for loop.

In both of these the error can be found by using the same traces that were used with the convergence loop. If the number of passes is zero, then clearly the loop never ran. If the number of passes is different than the number of elements that you thought were in the list then the loop is showing an off by one error.

The remedy is to review the three limits for the list - the starting value, the stopping value, and the step size. If the starting value is larger than the stopping value and the step is not negative, or it is missing, then the list will be empty and the loop will not run. The same would happen if the starting value is less than the stopping value but the step size is negative. There are no other ways to make an empty list and thus a for loop that never runs.

The off by one error also involves the three limits. The most common is that the step size is incorrect. Recalculate the number of elements - a formula for this will be presented in Chapter 10. If the number of elements is not the same as the number of passes that are needed in the loop, then reevaluate the step size.

Number of Elements
The number of elements in a list can be calculated using
n = floor((stop - start) ./ step) + 1;

It is also possible that the stopping value is incorrect. Remember that the list will end at or below the stopping value. It is not uncommon to erroneously include the stopping value as an element when it is in fact not included.

9.8 Summary

Repetition is the process in programming that separates the computer from the user. A person can perform even the most complex calculation with a pencil and paper. But what the person cannot do is repeat this calculation millions of times. A computer can perform such repeated calculations by using repetition.

There are two types of repetition, a convergence loop and an iterative loop. Convergence is done using a *while loop*. In the while loop a logical test is performed to determine if the loop should run. If the result of the test it true then a pass is made through the block of code that forms the loop. Once the pass is completed the test is repeated. This process continues until

the logical test returns false. The loop is described as a convergence loop because the process - test, run, update - is repeated until some value converges to the desired result.

Iteration is done with a *for* loop. Unlike convergence, in a for loop the program creates a list, or vector, of values. With this is a variable known as an iterator. The iterator is assigned the value of the first element of the list and then a pass is made through the loop. At the completion of the pass the iterator is assigned the second element and another pass is made. This process is repeated until each element in the list has been accessed.

While repetition enables to computers to process massive amounts of data they do come with an issue of time. A single loop will normally operate in linear time, while two loops nested one inside of the other will be quadratic. For large amounts of data the polynomial time needed to process the data using multiple nested loops can make an impact on the efficiency of a program. If a loop can be replaced by a single calculation then it is possible to reduce the complexity to a constant time algorithm.

In addition to the complexity issues, loops can fail in three other ways. In each it is a matter of an error in the programming. These are that the loop fails to run, the infinite loop, and the off by one error. In each case adding a trace to the loop can assist in identifying the error if exists and directing you to the appropriate fix.

Vectors 10

Smart data structures and dumb code works a lot better than the other way around.
Eric S. Raymond - Open Source Software Developer

Up to now, if we wanted to use a value in a program we had to create a variable. It is a straightforward one to one relationship, one number - one variable, But what if we do not know when we write the program how many values are going to be needed? We could use an accumulator and a single variable multiple times, but then we lose knowledge of the individual values. And what if we need them?

You want to calculate the mean of a set of observations. Easy enough, if you have three observations then

mean = (x1 + x2 + x3) ./ 3;
Since you have three values you create three variables, ***x1****,* ***x2****, and* ***x3****. But if you need four you will have to change the code.*

An alternative is to use a while loop

```
sum = 0.0;
count = 0;
while(count < n)
   sum = sum + x;
   count = count + 1;
end
mean = sum ./ n;
```

The while loop approach eliminates the need to know the number of observations, but you lose the individual values. This becomes an issue if you want to calculate the standard deviation.

What is needed is a method of data management that

- can store multiple values
- does not limit the number of values (can be one or one million)
- allows access to the individual values
- can be handled in the program like a single variable

The solution is to use a type of data structure called an *array* or a *vector*.

10.1 Manipulating Data in Vectors

Vector
A vector is a data structure that contains a group of elements.

The most common and widely used data structure is the one-dimensional *array* called a *vector*. It is similar to a variable but with an important difference - it can hold multiple pieces of data.

A vector consists of a set of *elements*. An element is the individual the data item that it being stored in the vector. Each element has an *index* starting at 1.

Warning
A vector can be used to store numerical values or characters, but not both.

If a program has a vector called **vec**, then **vec(k)** is the k^{th} element in the vector. k is the index of that element.

There are two types of vectors; row vectors and column vectors. Thinking of this in terms of matrix dimensions a row vector will have one row and n columns or a $1xn$ matrix of elements. A column vector will have m rows and only one column. This would make it an $mx1$ matrix of elements.

10.1.1 Creating a new vector

Vectors are only as useful as their data. But before they are used, they must be created. If a vector were a variable then the variable would be *declared*. But vectors are not variables in the traditional sense; they are an *object*. As we create an *instance* of an object. This process is often *instantiating* and object, or in this case *instantiating* a vector.

Instance
In object oriented programming an instance is an occurrence of the object. Creating this instance is called *instantiating*.

Vectors can be created in three ways; as an empty vector, by enumerating the individual elements, or by creating a range of data.

Creating an empty vector

The more common means of instantiating a vector consist of creating it by filling it with data. But there are many times when you may want a vector that that begins its lifetime with no elements. This is an *empty vector*.

```
1 %  An empty vector is created by assigning the [] to it
2 vector = [];
```

Figure 10.1: Syntax For Creating an Empty Vector

Once instantiated, an empty vector can be used in the same way as a vector that contains data.

Entering data by enumerating the elements

If the vector has only a few elements it can be created by enumerating; listing the individual elements in the vector separated by commas or spaces.

Enumeration
Elements are entered into a vector as a finite comma or semicolon delimited list.

```
%  Each element is entered in a comma delimited list
rowVector = [e_1, e_2, e_3, ... , e_n];
```

Figure 10.2: Syntax For Enumeration of a Row Vector

To create a column vector the commas are replaced with semicolons.

```
%  Each element is entered in a semicolon delimited list
columnVector = [e_1; e_2; e_3; ... ; e_m];
```

Figure 10.3: Syntax For Enumeration of a Column Vector

Example 10.4 demonstrates creating a row vector called **dataRow** with eight numerical elements, and another called **dataColumn** with five, on the command line.

```
>> %  Row vector - each element is comma delimited list
>> dataRow = [7, 5, 2, 9, 6, 8, 0, 4]
dataRow =
     7   5   2   9   6   8   0   4

>> %  Column vector - each element is semicolin
    delimited list
>> dataColumn = [3; 8; 1; 0; 5]
dataColumn =
     3
     8
     1
     0
     5
>>
```

Note
Row vectors and column vectors behave similarly. The distinction is minor now but will become important when they are used with matrices. Unless you specifically need one or the other it will not normally matter which you use, but you should be consistent. If you use row vectors then stay with row vectors. If you use column vectors then stay with column vectors. Avoid mixing the two in a single operation.

Figure 10.4: Row and Column Vectors with Eight Elements

Enumeration is the preferred choice if you just need to enter a small set of values that are not in particular order. But what if you need to create a vector with hundreds or thousands of values?

Entering the elements in a vector as a range

If the data is an increasing or decreasing series with a constant distance between points then you can create the vector as a range of values.

Figure 10.5: Syntax For Indicating the Range of Elements in a Row Vector

```
%  Elements are determined by a range of values
rangeVector = [start:step:stop];
```

In the range approach, you enter three colon delimited values; the starting point, the step size - the distance between each point, and the stopping value. While the first element of the vector will the starting value, the stopping value is a *do not exceed* value. This means that the final element will be at or below the stopping value but never above it.

Note
The final value in a vector will not pass the **stop** value.

The **step** value is optional. If it is omitted then it defaults to one. A common use of this - as it was in the **for** loop - is to create a vector with a set of values $1, 2, 3, \cdots, n$ where n is set in the program.

```
>> %  Row vector - unit step from one to eight
>> dataRow = [1:8]
dataRow =
     1   2   3   4   5   6   7   8

>> dataRow = [3.2:6.7]
dataRow =
     3.2   4.2   5.2   6.2

>>
```

Figure 10.6: Vector with Range from 1 to 8

Of course the range can start or stop at any values, positive or negative. If the **start** < **stop** then the elements will be increasing. But if **start** > **stop**, and **step** is omitted an empty vector will be created.

```
>> %  Row vector - increasing range
>> incVector = [-4:5]
incVector =
     -4   -3   -2   -1   0   1   2   3  4  5

>> %  Row vector - decreasing range
>> decVector = [6:-2]
decVector = [](1x0)

>>
```

Figure 10.7: Increasing and Decreasing Range of Elements in a Vector

Warning
If **start** > **stop** and **step** is omitted an empty vector will be created. No warning or error will be given.

The empty vector, as in example 10.9 is created and is accessible. It is a vector like any other it just does not contain any elements. The notation **(**$1x0$**)** indicates that it is a row vector; one row with zero columns.

If **start** $\leq$ **stop** $<$ **start** $+ 1$ then the vector will contain only a single element. This also covers the case where **start** $=$ **stop** (example 10.8).

```
>> %  stop = start
>> scalar = [2:2]
scalar =
     2

>> %  stop < start + 1
>> scalar = [5:5.7]
scalar =
     5

>>
```

Figure 10.8: Creation of a Scalar When **stop** $\leq$ **stop** $+ 1$

When the **step** is set the vector is created where each element is **step** away from the previous element. The final element will not always be **stop**. As mentioned previously, **stop** is a *do not exceed* value. This means that if adding an additional **step** to value results in an element that is beyond **stop** then the vector stops with lower value.

```
>> %  Row vector - increasing range
>> stepVector = [2:0.7:6]
stepVector =
     2.7   3.4   4.1  4.8   5.5

>> %  Row vector - decreasing range
>> decVector = [6:-1.3:-2]
decVector =
     6  4.7  3.4  2.1  0.8  -.5  -1.8
>>
```

Figure 10.9: Increasing and Decreasing Range of Elements in a Vector

The elements are created using the recursive formula

$$\begin{aligned} a_k &= a_{k-1} + \textbf{step} \\ a_1 &= \textbf{start} \\ k &= 2, 3, \ldots, [[1 + (\textbf{stop} - \textbf{start})/\textbf{step}]] \end{aligned} \tag{10.1}$$

Greatest Integer Funcdtion
The function $[[x]]$ is the greatest integer function (GIF). It is the largest integer that is less than or equal to x. In MatLab the GIF can be called using **floor(x)**.

The recursive formula shows that elements will be created in both increasing series, **step** > 0, and decreasing series, **step** < 0. But if **step** is in contradiction with the direction of **start** and **stop** then the recursive series fails. In this case you will still create a vector, but it will be empty.

Warning
While vectors can be created that are both increasing and decreasing, the sign of **step** must correspond to the whether the vector elements are increasing or decreasing. If they do not, the vector will

The range method can only be used to create row vectors. If you need to create a column vector then you do so in two steps. First you create the row vector then you take the matrix transpose using the transpose operator (**.'**). This will transform the row vector into a column vector.

```
>> %  Create the range then take transpose
>> colVector = [0:4].'
colVector =
     0
     1
     2
     3
     4

>>
```

Figure 10.10: Creating a Column Vector Using the Range of Elements

Assigning elements individually

Creating a vector by either enumerating all of the elements or by setting up a range of values is a common method if the values of the elements are know prior to running the program. But there are often times when you need to enter the value of an element one element at a time during the runtime. You can do this by assigning a value to an individual element.

```
%  Syntax For Entering Individual Elements
vector(k) = [e];
```

Figure 10.11: Syntax for Creating a Vector by Assigning Individual Elements

An example of this is shown in figure 10.12 where a new vector is created by first assigning a value to the first element; then the second, and the third, and so on.

```
>> %  Enter values into the individual elements by their
     index
>> vector(1) = 3;
>> vector(2) = 5;
>> vector(3) = 2;
>> vector(4) = 7
vector =
     3  5  2  7

>>
```

Figure 10.12: Entering Individual Elements

This is a common approach for when the user will be

entering the data through an input prompt as demonstrated in example 10.13.

```
function driver( )
% DRIVER driver( ) is the main or
      driver function

  % Create an empty vector
  X = [ ];
  % Create a counter and an input
      variable
  count = 0;
  x = 0;

  %  Enter data using a while
     loop
  %  Use a sentinel value (< 0)
     to stop
  fprintf('Enter each value\n');
  fprint('(negative to quit)\n');

  while(k > 0)
     k = input('Next value: ');
     if(x > 0)
        count = count + 1;
        X(count) = x;
     end
  end
  %  Print the vector using a for
      loop
  fprintf('\n X = ');
  for k = [1:count]
     fprintf('%d: x = %0.2f    ',
     k, X(k));
  end
  %  Print a new line
  fprintf('\n ', k, X(k));

end
```

```
>> driver( )
Enter each value
(negative to quit)
Next value: 7
Next value: 3
Next value: 6
Next value: -5

X = 7.00     3.00
      6.00

>>
```

Figure 10.13: Filling a Vector Using a **while** Loop and a Sentinel Value

The example also shows a means of printing the elements of a vector. Since each element can be accessed by its index, a **for** loop that iterates through a list of the vector indices will also access each element.

There is no requirement that the elements be entered in order. If the index of an element that is being entered is beyond the final element of the current vector, or not 1 for a new vector, the vector that you create will be filled with zeros up to the

Note
If the index exceeds the current size of the vector, the vector will be expanded and the missing elements will be assigned the value 0.

index specified. This will also create a new vector if one does not already exist.

```
>> %  Enter individual element
>> vector(4) = 3;
vector =
     0  0  0  3

>> vector(7) = 8;
vector =
     0  0  0  3  0  0  8

>>
```

Figure 10.14: Buffering Vector Elements with Zeros

This technique can be useful when creating a vector of all zeros - although we will later see another way. Assign a value of 0 to what is to be the last element of a vector. It will be set to 0 as will all of the elements that come before it.

```
>> %  Enter individual element
>> zeroVec(7) = 0;
zeroVec =
     0  0  0  0  0  0  0

>>
```

Figure 10.15: Creating a Zero Vector

10.1.2 Reassignment of Elements in a Vector

Mutable
An object if mutable if their components can be changed after they have been assigned.

Vectors are *mutable* - a term that simply means that the individual elements can be reassigned. Changing individual elements is a matter of reassigning a value to a particular element. This can be done individually - one element at time - or by range.

Assigning a scalar to a single element

The syntax for changing a single element is shown in figure 10.16.

```
%  Assigning a scalar to an element by indicating the
    index, k
vector(k) = [x];
```

Figure 10.16: Syntax for Changing an Individual Element of a Vector

When assigning data to a single element of a vector the index of the element is within a set of parentheses, **( )**, while the new element can - but is required to - be within a set of

square brackets **[ ]**. This creates a clear demarcation between the data in the element and location or index of that element. The sample code in figure 10.17 demonstrates this.

```
>> X = [6 3 8 1 4];
>> X(2) = [5]
X =
   6 5 8 1 4
>>
```

Figure 10.17: Changing an individual element of a vector

Warning
When reassigning an individual element or a range of elements, the index values are indicated with the vector inside of the parentheses. The values of the elements that will be stored in the vector are collected inside of square brackets.

Assigning a vector to a range of elements

It is also possible to reassign a range of elements using a single command. It is very similar to an assignment to a single element, but the index is replaced with a range of indices, and the scalar value for the element is replaced with a vector. The syntax for this is shown in example 10.18.

```
%  Assigning a vector to a range indicated by start:step
    :stop
vector(start:step:stop) = [e_1 e_2 ... e_n];
```

Figure 10.18: Syntax for Changing a Range of Elements of a Vector

There are several items that need clarification in how updating a range of elements is done.

- Because the range corresponds to the index values in the vector, the values of the range must be integers.
- The minimum value for the range is 1.
- If the step is excluded then it defaults to 1.
- The range can be increasing or decreasing, but if it is decreasing then the step must be included.
- If the length of the range of elements to be assigned in the range on the left of the assignment operator is n, then the length of the vector with the data to be assigned must be either n or one.

In this sample code, a range of three is indicated by the **(2:4)** and the vector on the right hand side has three elements. Thus the element values $3, 8, 1$ are reassigned to $5, 9, 0$.

What if the right hand side vector has a length of one - that is it only contains a single element? Then the single element is expanded to the same length of the vector using what is known as *scalar expansion*. This means that the scalar is repeated the number of times that are needed to make it appear as a vector with all elements the same.

Scalar Expansion
If a vector is being assigned a scalar to a range of elements, the scalar is automatically repeated using scalar expansion until it is the same length as the vector.

```
>> X = [6 3 8 1 4];
>> X(2:4) = [5 9 0]
X =
   6 5 9 0 4

>>
```

Figure 10.19: Assigning a vector of elements to a subrange of elements of another vector

```
>> X = [6 5 9 2 4 1 8 7 3];
>> X(3:5) = [3]
X =
   6 5 3 3 3 1 8 7 3

>>
```

Figure 10.20: Assigning a scalar to a range of elements in a vector using scalar expansion

The **step** parameter makes it possible to assign data to noncontiguous elements of the vector. For example, if you need to change every third element to 0 then you set **step** to 3 as in figure 10.21.

```
>> X = [6 5 9 2 4 1 8 7 3];
>> X(1:3:9) = [0]
X =
   0 5 9 0 4 1 0 7 3

>>
```

Figure 10.21: Using **step** to reassign data to noncontiguous elements

This will also work *right to left* - a decreasing range - as long as **start** > **stop** and **step** < 0. As always **start**, **step**, and **stop** must be integers.

```
>> X = [6 5 9 2 4 1 8 7 3];
>> X(9:-4:1) = [0]
X =
   0 5 9 2 0 1 8 7 0

>>
```

Figure 10.22: Using a negative **step** to reassign from right to left

In addition to mutability - the ability to reassign values to the elements of a vector - the length of the vector in terms of the number of elements will increase and decrease as elements are added or removed. This means that vectors are *dynamic*.

10.1.3 Dynamic Vectors

We have seen that we can create vectors and add elements to that vector by assigning a value to an individual element. If the vector does not exist then it is created with the first element assignment. If it does, and the index is beyond the last element of the vector, then the vector is expanded to include the new element. As we have seen the expansion may include the creation of additional elements of which each is assigned the value 0. This demonstrates that vectors are dynamic - they can grow with the addition of data. We also see that they can shrink as elements are removed from the vector.

Dynamic Vectors
A vector that can have data added to it and thus increase the number of elements, or data deleted decreasing the number of elements is a dynamic vector.

Vectors, being dynamic, can increase in length by adding elements or decreased by deleting elements.

Appending elements onto a vector

We have already seen an example of expanding a vector; assigning a value to an element beyond the current end of the vector. Examples of this are in figures 10.11 and 10.14. In both cases this is known as *appending* - adding data to then end of a list or a vector.

Appending
Appending data consists of adding elements to the end of a vector, thus increasing the length of the vector.

```
1 %  Appending elements after the end of a vector of
      length n
2 %  Note m > n and is an integer
3 vector(m) = [e_1 e_2 ... e_k];
```

Figure 10.23: Syntax for Appending Elements onto a Vector

Appending data is not limited to single elements. Vectors can be appended onto the tail of a vector as long as the range of the index values matches the number of elements in the vector to be appended. As with scalars, if you append a vector at a point beyond the current final element then the additional elements will be created with the value 0. The **step** parameter can also be set so that the vector being appended is not contiguous but instead separated by elements with the value 0.

Inserting elements within a vector

It is also possible to add elements to the head of a vector - the beginning - or the middle. Unlike appending this is known as *insertion*.

```
>> X = [3 7];
>> % Append a vector onto the tail
>> X(3:4) = [5 1]
X =
   3 7 5 1

>> % Append a vector beyond the tail with a step value
>> X(7:2:9) = [6 8]
X =
   3 7 5 1 0 0 6 0 8

>> % Append a scalar with a repeat of the value
>> X(10:2:12) = [2]
X =
   3 7 5 1 0 0 6 0 8 2 0 2

>>
```

Figure 10.24: Appending a vector onto the end of another vector

There is a slight difference in the syntax when inserting a vector at the head or tail than there is when inserting it within the middle of a different vector.

```
%  Inserting elements at the head of a vector
vector = [ [e_1 e_2 ... e_k] vector];

%  Inserting elements at the tail of a vector
vector = [ vector [e_1 e_2 ... e_k] ];
```

Figure 10.25: Syntax for Inserting Elements at the Head or Tail of a Vector

Insertion consists of creating a new vector consisting of the current vector and the scalar or vector to be inserted. Once created this new vector is assigned to the current vector, replacing it.

The difference that occurs when inserting within the vector is that the original vector must be split at the insertion point. The portion that is to remain in the front of the insertion point is indicated by its range while the remaining subvector is placed after the inserted vector and is indicated by its *original* range.

Warning
When inserting a vector into the middle, the ranges that are indicated are those of the current vector before the insertion.

```
%  Inserting elements after the mth element of a vector
    of length n
%  Note 1 <= m <= n and is an integer
vector = [vector(1:m) [e_1 e_2 ... e_k] vector(m+1:n)];
```

Figure 10.26: Syntax for Inserting Elements into a Vector

In figure 10.27 the second insertion takes place at index 4. Thus the first three elements of the current vector are placed

in the new, temporary, vector. After the vector that is to be inserted, the remaining elements of the original vector are placed at the tail. Since the remaining vectors started at the old index of 4 the are indicated with the old index value despite now being at index 6.

```
>> X = [3 7];
>> % Insert a vector at the head
>> X = [[5 1] X]
X =
   5 1 3 7

>> % Insert a vector in the middle
>> X = [X(1:3) [6 8] X(4)]
X =
   5 1 3 6 8 7

>>

>> % Insert a vector at the tail
>> X = [X [4 2]]
X =
   5 1 3 6 8 7 4 2

>>
```

Figure 10.27: Inserting a vector onto the head and in the middle of another vector

Inserting elements can be implemented individually - as a single element - or as an additional vector. With the insertion approach a new - albeit unnamed vector is temporarily created. Once the insertion is completed the temporary vector is assigned to the original vector - replacing the old with the new.

The number of elements in a vector can be increased by insertion anywhere in the vector - head, middle, or tail - and also by appending onto the tail.

But in keeping with the dynamic nature of vectors - if elements can be added they should be able to be removed. Thus elements can be deleted as well.

Note
Appending elements onto the tail of a vector allows the ability to buffer missing elements with zeros and also the ability to set a **step** value to have the appended vector skip elements by adding in zero values. Inserting does not. Any vector or scalar inserted will done by creating a *gap* in the original vector and filling that gap with the new elements.

Deleting elements from a vector

Deleting an element or a set of elements from a vector is accomplished in the same way if the elements are at the head, the middle, or the tail. In fact it is not so much a matter of deleting the elements as it is reassigning them with ... nothing.

To delete an element, a range of elements, or selected elements you reassign them with the empty vector, **[ ]**.

```
%  Deleting elements from an interval of a vector
vector(start:step:stop) = [];
```

Figure 10.28: Syntax for Deleting Elements from a Vector

In its most basic - deleting a single element - you set that element to the empty vector. After doing so will realign the indices for the vector so that all of those after the deleted element will shift one to the left.

```
>> X = [1:15];
>> % Delete the fifth element
>> X(5) = []
X =
   1 2 3 4 6 7 8 9 10 11 12 13 14 15

>> % Delete the current 7th through 9th element
>> X(7:9) = []
X =
   1 2 3 4 6 7 11 12 13 14 15

>>

>> % Delete every third element of those remaining
>> X(3:3:length(X)) = []
X =
   1 2 4 6 11 13 14

>>
```

Figure 10.29: Deleting an Element or a Range of Elements from a vector

Appending and inserting elements is reversible. As long as you retain the length of the original vector or the location of the inserted elements you can then delete them. But not so with deletion. Once elements are deleted they are gone.

10.2 Operations With Vectors

Vectors provide a means of storing and moving data through a program with only a single variable. This includes passing vectors to a function and returning them from functions. Whether the operations with vector take place within a function or outside of one, the computations can take advantage of their multiple data elements.

These calculations happen in two ways; as one-dimensional matrices or as individual elements. The first will follow the

properties of matrix operations while the latter is done *element-wise*.

10.2.1 Element-Wise Operations

The most common use of vectors is in element-wise operations. In this type of operation computations are performed between a vector and a scalar, or two vectors of the same type and size.

Element-wise operations with vectors are operations in which the computation is performed between the elements in the first vector with the elements in the same location of the second. Thus if you have two vectors, **X** and **Y** then the first element of **X** will be matched with the first element of **Y**. The second of each will be paired, and the third, and so forth.

If we use $\odot$ to indicate any element-wise operation then

$$\begin{aligned} Z &= [a_1, a_2, \ldots, a_n] \odot [b_1, b_2, \ldots, b_n] \\ &= [a_1 \odot b_1, a_2 \odot b_2, \ldots, a_n \odot b_n] \end{aligned} \tag{10.2}$$

Warning
The sample code in figure 10.30 is included to show how an element-wise operation could be written. It is included simply as a chance to *look under the hood*. You should not actually write your vector operations using **for** loops but instead use the element-wise operators.

While element-wise operations are built in as callable operators, it is possible to write code using a single for loop that perform an element-wise operation on two vectors - or a vector and a scalar. A demonstration of this in figure 10.30.

Although any of the element-wise operations can be written using a **for**, loops, especially iterative loops, are considered slow when using an interpreted language. As such they should be avoided. Instead MatLab provides built-in operators for the five element-wise operations between vectors, addition **+**, subtraction **-**, multiplication **.***, division **./**, and exponentiation **.**$\wedge$.

Element-wise calculations are the standard when performing any calculation with vectors as examples 10.32 and 10.33 show.

Note
A question about calculations is often asked. "Why use **.***, **./**, and **.**$\wedge$ instead of the the commonly accepted *, /, and $\wedge$? The more common operators are used for matrix operations. The *dot operator* is used to differentiated matrix from element-wise operations.

If the operations are between a vector and a scalar then the scalar expansion is performed so that the scalar is the same length as the vector. From there the usual element-wise operations are done. Of course the scalar expansion is never seen.

```
function driver( )
% DRIVER driver( ) is the main or driver function

  %  Set the length
  n = 8;
  %  Create two vectors
  X = [3, 4, -3, 6, 8, -7, -1, 9];
  Y = [6, 1, 5, 9, 7, 2, -3, 4];

  % Start a for loop to add the elements of the
  % two vectors. The iterator a list from 1 to n
  for k = 1:n
     %  Add the corresponding elements
     Z(k) = X(k) + Y(k);
  end

  %  Show the elements of Z
  Z

end
```

```
>> driver( )
Z =

  9  5  2  15  15  -5  -4  13

>>
```

Figure 10.30: Adding the elements of two vectors using a for loop

```
%  Addition
vectorSum = vec_1 + vec_2;
%  Subtraction
vectorDifference = vec_1 - vec_2;
%  Product
vectorProduct = vec_1 .* vec_2;
%  Quotient
vectorQuotient = vec_1 ./ vec_2;
%  Power
vectorPower = vec_1 .^ vec_2;
```

Figure 10.31: Syntax for the Five Element-Wise Operations

10.2.2 Matrix Operations with Vectors

While we commonly think of vectors as a data structure to store and manipulate multiple elements, they are at the most basic a one-dimensional matrix. A row vector is a $1xn$ matrix and a column vector is an $mx1$ matrix.

Matrix operations with a vector and a scalar are the same regardless of whether it is done using element-wise or matrix calculation. As such, if you are adding, subtracting,

Note
Using the operator with the dot or without when adding, subtracting, or multiplying a vector and scalar will not change the result. A matter of consistency is to choose the operator based upon whether the operations are meant to be matrix or element-wise. If matrix operations then do not use the dot. If element-wise then use the dot.

A question is often asked about calculations. "Why use **.***,* **./***, and* **.^** *instead of the same operator without the dot? After all, it works just fine with scalars."*

The answer is that all operations were originally meant to be used with matrices in matrix operations. Because of this the common operators for multiplication, division, and exponentiation were assigned to their matrix calculation. And yes, it does work for scalars because a scalar is a $1x1$ *matrix and for scalars the matrix operations and the element-wise operations are the same. But they are not the same for vectors or other matrices.*

This is also the reason that you do not need the dot for **+** *or* **-***. Matrix addition and subtraction are the same as element-wise addition and subtraction. Since there is no difference there is no need for the dot.*

```
function driver( )
% DRIVER driver( ) is the main or
      driver function

    % Create vectors
    X = [6, 3, 9, 7]
    Y = [2, 0, 3, 1]

    %  Addition
    A = X + Y
    %  Subtraction
    S = X - Y
    %  Multiplication
    M = X .* Y
    %  Division
    D = X ./ Y
    %  Exponentiation
    E = X .^ Y
end
```

```
>> driver( )

X =
   6   3   9   7
Y =
   2   0   3   1
A =
   8   3   12   8
S =
   4   3   6   6
M =
   12   0   27   7
D =
   3   inf   3   7
E =
   36   1   729   7

>>
```

Figure 10.32: Demonstrating Element-Wise Operations with Two Vectors

multiplying a vector and a scalar the choice of operator, with the dot or without, does not matter.

If the row vector is multiplied to the column vector then each must have the same dimension (number of elements). But the dimensions of the two vectors do not have to be the same if it is the column vector multiplying the row vector. Recall that when multiplying matrices the columns of the first must

```
function driver( )
% DRIVER driver( ) is the main or
     driver function

    % Create a scalar and a
    vector
    a = 4;
    X = [2, 0, 3, 1];

    %  Addition
    A = a + X
    %  Subtraction
    S = a - X
    %  Multiplication
    M = a .* X
    %  Division - Scalar
    Denominator
    DSD = a ./ X
    %  Division - Vector
    Denominator
    DVD = a ./ X
    %  Exponentiation - Scalar
    Base
    ESB = a .^ X
    %  Exponentiation - Vector
    Base
    EVB = X .^ a
end
```

```
>> driver( )

A =
   6   4   7   5
S =
   2  -4   1  3
M =
   8   0  12   4
DSD =
   0.5   0   0.75
     0.25
DVD =
   2   inf   1.33
     4
ESB =
   16   1   64   4
EVB =
   16   0   81   1

>>
```

Figure 10.33: Demonstrating Element-Wise Operations with Vector and Scalar

match the rows of the second. Thus for two vectors

$$\begin{aligned}(1xn) * (nx1) &= 1x1 \quad \text{(scalar)} \\ (nx1) * (1xm) &= nxm \quad \text{(matrix)}\end{aligned} \tag{10.3}$$

The first product, the scalar, is the *dot product* or *scalar product*. It is a common calculation in engineering for calculating the magnitude of a vector or the angle between two vectors.

Dot Product
The dot product, or *scalar product* is the matrix product of a row vector and a column vector of the same length. It is calculated as the sum of the products of each matching element. It can be used to determine the angle between two vectors.

The magnitude of a vector is its length. It is calculated by

$$|u| = \sqrt{u \cdot u^T} \tag{10.4}$$

The magnitude can be calculated as a single line code - that can be run on the command line - in which a vector is multiplied using the matrix product with its own transpose (figure 10.34).

```
>> X = [4, -3, 6];
>> % Calculate the magnitude as the matrix product with
    its transpose
>> magX = sqrt(X * X.')
magX =
     7.8102

>>
```

Figure 10.34: Dot product to calculate the magnitude of a vector

The dot product can be used to calculate the angle between two vectors.

$$\begin{aligned} u \cdot v &= |u|\,|v| \cos(\theta) \\ \theta &= \cos^{-1}\left(\frac{u \cdot v}{|u|\,|v|}\right) \end{aligned} \tag{10.5}$$

there are two different vectors then the dot product can be used to calculate the The sample code in figure 10.35 demonstrates using the vector product to calculate the angle between two vectors in 3-space.

Vectors calculations can be made both element-wise and matrix wise. This puts the vector on the same level as scalars in terms of calculation, but can vectors be compared to one another? And what would comparing two vectors even mean?

```
function driver( )
% DRIVER driver( ) is the main or driver function

  % Create two vectors
  X = [4, -3, 6];
  Y = [1, 5, -2];

  % Calculate the magnitude of each of the two vectors
  % Need to take the transpose of the second vector
  magX = sqrt(X * X.');
  magY = sqrt(Y * Y.');

  % Calculate the dot product by multiplying two vectors
  % Need to take the transpose of the second vector
  dotProduct = X * Y.';

  % Calculate the angle between the vectors
  angle = acosd(dotProduct ./ (magX .* maxY);

  % Print the results
  fprintf('Angle: %0.2f degrees\n', angle);

end
```

```
>> driver( )

Angle: 122.52 degrees

>>
```

Figure 10.35: Dot product to calculate the angle between two vectors

10.3 Vectors, Relational Operators, and Comparison

Is it possible for a vector to be smaller, or larger than a scalar? How about two vectors? Can one vector be smaller than the second. In a basic sense - no. The magnitude of each vector can be compared or the magnitude of a vector can be compared to a scalar, but not the entire vector.

Comparing the magnitude of two vectors - or the magnitude of a vector to a scalar is an application of the dot product. As an example, figure 10.36 demonstrates how to determine if the magnitude of one vector is larger than than other.

Since in engineering we often use vectors in their traditional sense of a ray with direction and magnitude, this application would be used to determine which vector is longer. But this application is not a comparison of vectors as much as it is a comparison of scalars - which we have done before.

```
function driver( )
% DRIVER driver( ) is the main or driver function

  % Create two vectors
  X = [4, -3, 6];
  Y = [1, 5, -2];

  % Calculate the magnitude of each of the two vectors
  % Need to take the transpose of the second vector
  magX = sqrt(X * X.');
  magY = sqrt(Y * Y.');

  % Compare the vector magnitudes
  if(magX > magY)
     fprintf('Vector X is larger than Vector Y\n');
  else
     fprintf('Vector X is not larger than Vector Y\n');
  else

end
```

```
>> driver( )

Vector X is larger than Vector Y

>>
```

Figure 10.36: Comparing the magnitude of two vectors

Comparing vectors as data structures is different from comparing them as geometrical vectors. In the data structure implementation vectors are compared element-wise. This means that if a relational operation between two vectors, **X** and **Y** will result in a third vector where each element indicates with true or false (1 or 0)

In this example, figure 10.37, each element is compared with the result being its logical value.

This same approach is used when comparing a vector to a scalar. In this matter the scalar is expanded to the same length of the vector and each element of the vector is then compared to the same value. This can be used, as in the example in figure 10.38, to see if all of the elements are above some minimum value. And if they are not, which elements fail the test.

Vector operations can be implemented both element-wise and matrix wise. They can also be compared element-wise to another vector or by scalar expansion to a single scalar. We have developed these operations to be performed outside of

Note
A useful addition to this would be know how many of the elements are zero or one. While we will later see several functions that will do this directly we could adapt the code to provide us the number of 1s and the number of 0s. Recall that the dot product of a vector with itself will return a sum of the squares of the elements. And since all of the elements are either 1 or 0 then the sum of squares will be the number of true values. If we then subtract each element from 1 and repeat it we will have the number of zeros.

```
function driver( )
% DRIVER driver( ) is the main or driver function

  % Create two vectors
  X = [3, 4, -3, 6, 8, -7, -1, 9];
  Y = [6, 1, 5, 9, 7, 2, -3, 4];

  % Compare the vectors element-wise
  % This will result in a vector of 0s an d1s
  TorF = (X < Y)

  % Number of ones
  numOnes = TorF * TorF.'
  % Number of zeros
  numZeros = (1-TorF) * (1-TorF).'

end
```

```
>> driver( )

TorF =

  1  0  1  1  0  1  0  0

numOnes =  4
numZeros =  4

>>
```

Figure 10.37: Comparing the elements of two vectors

functions, but in keeping with our goal of top-down design we need to be to hide them away within functions as well.

```
function driver( )
% DRIVER driver( ) is the main or driver function

  % Create two vectors
  X = [5, -5, 2, 9, -1, 3, 0, 6];
  a = 0;

  % Compare the vector to the scalar
  % This will result in a vector of 0s an d1s
  TorF = (X >= a)

  % Number of ones
  numOnes = TorF * TorF'
  % Number of zeros
  numZeros = (1-TorF) * (1-TorF).'

end
```

```
>> driver( )

TorF =

  1  0  1  1  0  1  1  1

numOnes =  6
numZeros =  2

>>
```

Figure 10.38: Comparing the elements of a vector with a scalar

10.4 Vectors and Functions

The relationship between vectors and functions are similar to those with scalars. Vectors can be passed as a parameter to a function, and they can be returned as an output from a function. But is also possible to have scalar inputs with vector outputs, vector inputs with vector outputs, and vector inputs with scalar outputs.

10.4.1 Functions that are passed vectors and return vectors

Most functions in which a vector is passed returns a vector consisting of the elements calculated using element-wise operations. This is the case for any of the computation functions like the trigonometry functions **sin**, **cos**, and **tan**. This is the same for the algebraic functions such as **log**, **exp**.

An example of a code in which a vector is passed to the **cos** function and the return vector is then part of an additional element-wise calculation is shown in figure 10.39.

```
>> % Create vector
>> X = [0, 30, 45, 60, 90]
>> %  Evaluate the cosine with angle in degrees
>> Y = cosd(X)
Y =
   1.00000   0.86603   0.70711   0.50000   0.00000

>>
```

Figure 10.39: Element-Wise Operations in Evaluating a Built-In Function

Element-wise calculations with a vector return are the common result of most local and anonymous functions.

```
>> % Create the anonymous function
>> f = @(x) x.^2 + 3.*x - 10;
>> X = [-3, -2, -1, 0, 1, 2, 3, 4];
>> % Evaluate the function
>> Y = f(X)
Y =
     -10 -12  -12  -10   -6    0    8   18

>>
```

Figure 10.40: Element-Wise Calculations in an Anonymous Function

It should not be assumed that functions that are passed and return vectors are simply computational. There are many applications in which you need to rearrange elements in a vector or sort them in ascending - lowest to highest - or descending - highest to lowest - order. These will be addressed in section 10.5.

10.4.2 Functions that are passed scalars and return vectors

Most of the functions with which we have dealt have been computational; pass scalar values as input parameters - calculate and returns another scalar. With vectors and element-wise operations nothing changes. We pass a vector or vectors to the function, and it uses them to calculate and return another function.

But there are functions that are passed individual scalars and these are used to create a vector.

The step size for a vector given the **start** and **stop** values with a set number of elements could be calculated in a function

Example: A common issue in creating a vector is determining the correct step size for the range. Say you want a vector with just five elements from 0 *to* 100. *Many people would erroneously say the* ***step*** *should be set at* 20 *but this would create a vector with six elements. Close, but six is not five.* ***step*** *should not have been set to* 20 *but instead to* 25.

and the function could then create and return the vector with the correct number of elements. The step size - or the number of elements - can be calculated using the formula

$$\textbf{step} = \frac{\textbf{stop} - \textbf{start}}{n - 1} \tag{10.6}$$

The number of elements is decreased by one, $n - 1$, because the divisor is not the number of elements but instead the number of steps. For any vector there will always be one less step than there are elements.

If needed, equation 10.6 can be solved for n to determine the number of elements.

$$n = \left\lfloor \left[\frac{\textbf{stop} - \textbf{start}}{\textbf{step}} + 1 \right] \right\rfloor \tag{10.7}$$

The formula in equation 10.6 can also be used to create a local function that creates a vector.

There is a built in function that does the same as the example function **V = makeLinearVector(start, stop, n)**. It is **V = linspace(start, stop, n)**.

In engineering it often occurs that we need a vector of values that spaced logarithmically, not linearly. For example, instead of a vector $\mathbf{V} = [0, 1, 2, \ldots, n]$, the vector needed is $\mathbf{V} = [10^0, 10^1, \ldots, 10^n]$. This can be done using the **makeLinearVector** function (or **linspace**) and the element-wise power operator in figure 10.42.

The sample function **makeLogVector** performs the same operation as calling the built-in function **VLog = logspace(start, stop, n);**

There are several functions similar to **linspace** and **logspace** but they are designed to return matrices. But they are useful enough in creating vectors to adapt them here. Three in particular are **zeros**, **ones**, and **rand**. So while we can use them as matrix functions and then force them to create

linspace(start, stop, n)
The sample code in figure 10.41 demonstrates how to create a vector from the overall range and the number of elements. This code does not include the necessary error checking - **stop** $\geq$ **start** and $n > 0$ and and also an integer, $n \in \mathbb{Z}$. While we could add this, there is already a built in function to do that for us, **linspace(start, stop, n)**

logspace(start, stop, n)
Much like **linspace** there is a built-in function to create the logarithmically space function. It is **logspace(start, stop, n)**.

Note
The built-in functions **linspace** and **logspace** perform the same actions, so why create local function? In this case it is a matter of generalization. The built-in function is convenient but the local function can be adapted to, for example, a binary growth function by changing **V = 10.^VLin** to **V = 2.^VLin**.

```
function V = makeLinearVector(start, stop, n)
%  MAKELINEARVECTOR V = makeLinearVector(start, stop, n) creates
%  a row vector with elements spaced linearly starting at start and stopping
%  at stop with n elements

   %  Calculate the step size
   step = (stop - start) ./ (n - 1);

   %  Create the vector
   V = [start:step:stop];

end
```

```
>> X = makeLinearVector(3, 17, 8)

X =

    3    5    7    9   11   13   15   17
>>
```

Figure 10.41: Creating a Linearly Spaced Vector

```
function v_log = make_log_vector(start, stop, n)
%  MAKE_LOG_VECTOR V = make_log_vector(start, stop, n) creates
%  a row vector starting at 10.^start and stopping at 10.^stop with n elements
%. logarithmically spaced

   % Create a linear vector
   v_lin = makeLinearVector(start, stop, n);

   %  Element-Wise Power Operator
   v_log = 10.^v_lin;

end
```

```
>> X = make_log_vector(0, 3, 8)

X =

      1.0000      5.6234     31.6228    177.8279
   1000.0000
>>
```

Figure 10.42: Creating a Logarithmically Spaced Vector

vectors, we could just write our own local functions that create the vector directly.

The first two are easy. We can set **start** = 0, and **stop** = n and then fill the elements with the value 0 or 1.

```
function zeroVector = makeZeroVector(n)
%  MAKEZEROVECTOR zeroVector = makeZeroVector(n) creates a row
%  vector starting of n elements all with the value 0

   % Set start to 1 and use n for stop and use scalar expansion to
   % fill all of the elements with zero
   zeroVector[1:n] = 0;

end
```

```
>> Z = makeZeroVector(7)

Z =

     0     0     0     0     0     0     0
>>
```

Figure 10.43: Creating a Zero Vector

While 0 and 1 are commonly used when creating an accumulator for sums or products, this technique can be used to create a vector with any starting value. For a different starting value, you change the assignment value in the function.

This type of function can actually be generalized for such as case by passing two input parameters to the function. The first is for the number of elements while the second is the value. If the user wants a default value, such as 0 then the function can use the **nargin** variable to overload it. An example of this is shown in figure 10.44.

10.4.3 Functions that are passed vectors and return scalars

There are a class of functions that when they are passed a vector of data they analyze the vector and return some value based upon that analysis.

Probably the most useful of this type of function is one that calculates the number of elements in the vector. A function that is passed a vector could then count the number of elements using a **for** loop that iterates through the vector each time updating an accumulator.

It makes little sense to add this local function to a program since it has already been written and is built-in. The built-in function is **length(X)**

Warning
MatLab provides a built-in function, **length**, that does the same operations as in the local function in figure 10.45. As such there is little reason to add the local function to a program. Instead call **n = length(X);** directly when you need to know the number of elements.

length
The built-in function, **length**, returns the number of elements in a

```
function valueVector = makeValueVector(n, a)
%  MAKEVALUEVECTOR valueVector = makeValueVector(n) creates a
%  row vector with n elements all with the value a

   %    Check nargin. If nargin == 1 then default to 0
   if((nargin < 1) | (nargin > 2))
      %  Exit with run time error for improper number of parameters
      error('Incorrect number of inputs to makeValueVector');
   elseif(nargin == 1)
      %  Set to the default of 0
      a = 0;
   end
   % Set start to 1 and use n for stop and use scalar expansion to
   % fill all of the elements with a
   valueVector[1:n] = a;

end
```

```
>> V = makeValueVector(7, 3)

V =

     3     3     3     3     3     3     3
>>
```

Figure 10.44: Creating a Vector with a single repeated value

While you would probably not write this function, it demonstrates how a function can be passed a vector and then analyze the vector returning a result. There are many such direct applications that could be written but because of the utility have already been included as built-in functions.

An example of this has already been demonstrated with the calculation of the *dot product*. This can be rewritten as a function; either a local function or an anonymous function.

```
function n = length(X)
% LENGTH n = length(X) counts the number of
% elements in a vector

  %  Set the accumulator to 0
  n = 0;

  % Use a for loop to iterate through the vector
  for k = X
     %  Update the accumulator
     n = n + 1;
  end

end
```

```
>> n = length([6 2 3 8 1 0 9 3])
n =

  8

>>
```

Figure 10.45: Function to return the number or elements in a vector

```
function dp = dotProduct(X, Y)
%  DOTPRODUCT dp = dotProduct(X, Y) calculates the dot
%  product of two vectors - it checks that the first
    vector is a row
%  and the second a column vector before doing the
    calculation

   %    Check nargin. If nargin == 1 then default to 0
   if((nargin < 1) | (nargin > 2))
      %  Exit with run time error for improper number of
     parameters
      error('Incorrect number of inputs to
    makeValueVector');
   elseif(nargin == 1)
      %  Set to the default of 0
      a = 0;
   end
   % Set start to 1 and use n for stop and use scalar
    expansion to
   % fill all of the elements with a
   valueVector[1:n] = a;

end
```

```
>> V = makeValueVector(7, 3)

V =

     3     3     3     3     3     3     3
>>
```

Figure 10.46: Dot Product Function

10.5 Sorting Elements in a Vector

If often occurs that the elements in a vector are not were we would like them to be. Perhaps two or more elements should be rearranged, or an entire vector needs to be sorted in either ascending or descending order. There is a built-in function to sort the elements of a vector in ascending order, but perhaps you need the vector in descending order. Or you only need several elements rearranged. In these cases you might want to customize the control of the elements.

10.5.1 Swapping elements in a vector

Sorting elements in a vector begins with being able to *swap* them. As an example, we want to swap the element at index k with the element at index j. This function, let's call it **swap**, will require three inputs; a vector, and the two indices for the elements to be swapped.

The swap function has multiple applications but probably the most common is in sorting the elements of a vector.

Note
A common error in a swap is omitting the intermediate *temp* variable. An analogy is to imagine a farmer with a pen of chickens and another with feed for the chickens to eat. He needs to swap the chickens and the feed. If he put the chickens directly in with the feed they would start to eat the feed. Similarly if he put the feed directly in with the chickens. Instead he needs a temporary third pen. He then transfers the chickens to the temporary pen. He then moves the feed to the now empty pen that previously was full of chickens. Once the feed has been moved he can then transfer the chickens over to their new pen - and then remove the temporary enclosure.

10.5.2 Sorting a vector

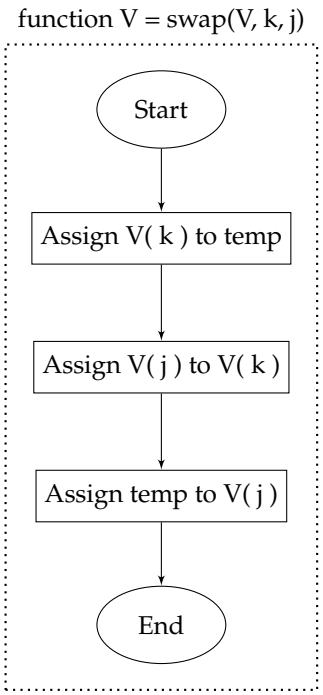
Vectors that are created using a range are already sorted; lowest to highest or highest to lowest. But if the vector was created by enumerating the elements or appending data, or if data is inserted into a vector it may no longer be sorted. There is a built-in function, **sort**, that will sort a vector in ascending order, but you may want to sort the data in descending order, or perhaps only sort every other element. As such there are many different types of sorts as well as techniques to sort the vectors.

A common, and easy to code - sort algorithm is called the *bubble sort*. This technique uses a pair of nested for loops. In this method the outer loop steps through the 1^{st} through $n - 1^{\text{st}}$ element of the vector. Each iteration starts the inner loop which compares the outer loop element to each element of the vector from the $k + 1^{\text{st}}$ to the n^{th} element. At each step if the elements are out of order then they are swapped - can be done with the **swap** function.

sort
The **sort** function is passed a vector, either a row vector or a column vector, and it returns the vector sorted in ascending order. The syntax is

```
V_sorted = sort(V_unsorted);
```

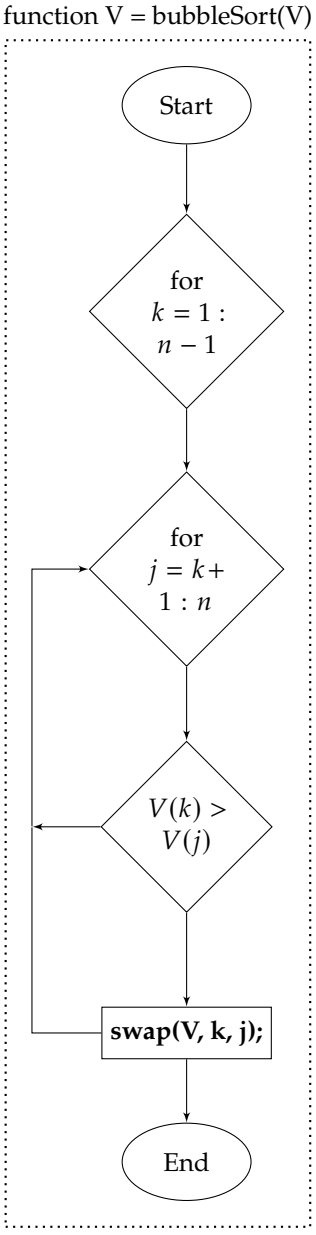
If the input and output vectors are the same the original will be replaced. But if they are different then there will be two instances of the vector - one sorted and one not.




```
1  function V = swap(V
       , k, j)
2  %  SWAP V = swap(V,
        k, m) swaps
       the
3  %  elements in
       index k of the
       vector
4  %  V with index j
       of the same
       vector
5
6     %  Copy first
       element to a
       temp
7     %  variable
8     temp = V(k);
9     %  Reassign the
       first element
10    %  with the
       value in the
       second
11    V(k) = V(j);
12    %  Reassign the
       second element
13    %  with the
       value in temp
14    V(j) = temp;
15
16 end
```

```
1  >> X = [1:5]
2  X =
3
4      1    2    3
          4    5
5
6  >> X = swap(X,
       2, 5)
7  X =
8
9      1    5    3
          4    2
10 >>
```

Figure 10.47: Function to swap elements




```
function V =
    bubbleSort(V)
%  BUBBLESORT V =
    bubbleSort(V)
    uses
%  a bubble sort
    algorithm to
    sort
%  the vector from
    lowest to
    highest

   %  Outer loop
   for k = 1:length
    (V) - 1
     %  Start
    inner loop
      for j = k+1:
    length(V)
         %  Check
    order
         if(V(k) >
    V(j))
             %  Out
    of order so
    swap
             V =
    swap(V, k, j);
         end %  End
     of if
      end  %  End
    of inner loop
   end  %  End of
    outer loop

end
```

```
>> X = [4, 1,
    5, 3, 2]
X =

    4    1
    5    3
    2

>> X =
    bubbleSort
    (X)
X =

    1    2
    3    4
    5
>>
```

Figure 10.48: Bubble sort function

Matrices 11

Index