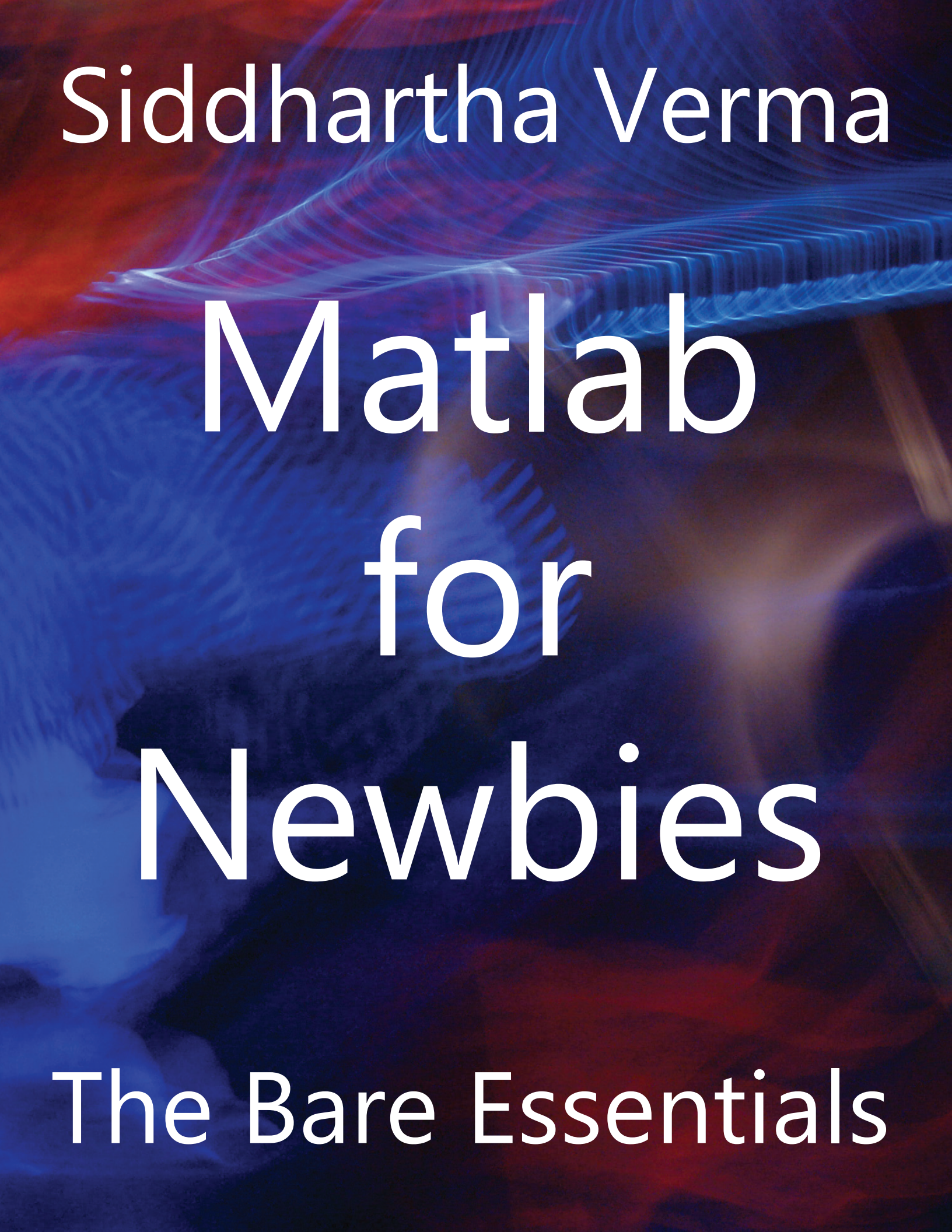

Siddhartha Verma
Matlab
for
Newbies
The Bare Essentials

Table of Contents

INTRODUCTION

Why should I learn programming in Matlab?

Since you picked up this book, I am assuming that you already have a good reason to consider using Matlab for its number-crunching prowess! No matter whether you are a biologist, a chemist, or an engineer, at some point in your work, you will need to perform quantitative analysis of raw data. Although programs like Microsoft Excel may seem like good, easy choices now, you will quickly realize that the simplicity that spreadsheet-software offers can quickly become more limiting than empowering.

If you are not convinced, do a quick image-search on the internet for terms like 'data analysis/visualization with Matlab', or something similar, then repeat the search for any spreadsheet-program that you might be considering. You will quickly realize that the capabilities of spreadsheet-software pale in comparison to what you could potentially do with Matlab.

> There are some alternatives to using Matlab that are quite powerful, for instance, 'GNU Octave' (https://www.gnu.org/software/octave/), which is an open-source twin of Matlab.

Matlab has a long history, is quite easy to use, and has an abundance of free help available online. If you study or work at a university, chances are that you already have access to a licensed copy. If you prefer personal use, you can buy a comparatively inexpensive home-use or student license. If you want a free option, GNU Octave is something you might want to look at. Most basic programs that you write for Matlab will run in Octave without any modification. In fact, most of the plots you will see in this book were generated with Octave.

The bottom line is that this book will guide your first steps in programming in Matlab, no matter whether you want to learn it for work, fun, or just to satisfy your curiosity!

What if I have never programmed before?

Then you are in the perfect place! This book was written specifically with you in mind. It can be daunting to deal with programming constructs if you have never encountered these before. In fact, I vividly remember feeling extremely confused and frustrated when I took my first 'Computer Science for Engineers' class during freshman year of college.

However, you will soon realize that getting comfortable with a few basic concepts will help you piece together simple instructions, to create sophisticated programs rather quickly. If you find that hard to digest, then consider the following. When studying engineering, it is natural for students to feel bombarded with complicated mathematical concepts; but most of the time, engineers use mathematics no more complicated than what we learn in high school.

Getting comfortable with the basics of programming in Matlab will be our main goal in this first of several segments that I hope to write. We will focus precisely on the things that you will need to get set up and running. You will be able to interpret simple code, and at least be able to understand what the code's author is trying to achieve.

I do experimental work. Why should I learn how to 'program'?

If your work is mostly experimental in nature, it is very natural to ask, "Why in the world will I ever need to program?" If you humor me for a minute, imagine that we are trying to sell your old, aging laptop to a researcher working in an experimental lab, and the year is 1960! It is natural for the researcher to be skeptical. After all, what use could they possibly have for an advanced 'computing machine'? Then, as you slowly show them all the fantastic things that they can do with it, you make them one of the happiest people on the planet!

Think of programming in similar terms: as a tool that will make your work much more effective and efficient. It will very quickly let you outgrow several limitations that using customized or pre-packaged software might impose on you. The return on investment in terms of increased productivity is enormous, even with just a few days' effort spent learning the basics.

On a different note, learning how to make the computer do repetitive grunt work for you does require some effort, especially in the beginning. But not having to deal with repetitive tasks will leave you with a lot more leisure time to do things that the computer can't, such as coming up with creative ideas.

Furthermore, basic knowledge of programming is a tremendous tool to add to your repertoire! It will make you stand out among your peers when you go for your dream interview.

Matlab vs. low-level languages

You might have considered using heavy duty, 'low-level' programming language like C,

C++ or Fortran for your work. However, If you are truly beginning programming as a newbie, the learning curve might be too harsh. It might take you months just to get up to speed in a 'low-level' language, as opposed to days for learning Matlab. If your eventual goal is to use low-level languages for mathematical tasks, but you don't yet have any experience with programming, then Matlab can be the perfect 'gateway programming language' for you.

You might be surprised to learn that Matlab is the first program that most researchers (including me) run to when we need to quickly test an idea, and most of the times even for in-depth data analysis. The productivity gap between using a 'high-level' language like Matlab (or Octave), and low-level languages (C, C++, Fortran) for routine tasks is enormous, even for those of us who are relatively fluent in using these low-level languages. We almost exclusively reserve the use of these low-level languages for extremely intensive computing tasks. Unless you get involved with computational research, it is unlikely that you will ever need them.

Thinking like a Computer

Before we start, there is one thing you must hammer into your brain: the computer usually starts with a 'clean sheet', and it can only perform the tasks that we instruct it to do. Any program/script we write instructs the computer to perform a set of tasks, in the particular order that we specify. The computer will do *nothing more*, and *nothing less* than what we ask it to do. This requires us to translate the ideas in our head into a coherent set of sentences that the computer will understand.

I remember that during the first couple of days of my freshman programming class, I was struggling to get used to the idea of 'translating' my thoughts into these set of instructions. Learning how to do this in a logical and structured manner will be one of our main goals of this book.

Is this Book Right For Me?

Finally, this book may not be right for you if you have already taken introductory courses in computing, and are very comfortable writing simple programs in Matlab, C,

C++, Fortran, or any of the several other programming languages. If you are interested in more advanced topics for programming in Matlab, you might consider looking at subsequent volumes that I plan to write in this series.

This book is intentionally short, to keep the focus tightly on the essentials, and filter out any 'fluff'. Hopefully, this will get you to the point where you can explore deeper concepts on your own, on one of the many freely available websites without feeling overwhelmed.

Summary

And now, with those words of wisdom out of the way, let us jump right into interacting with Matlab. Begin by typing the commands discussed in the following sections, at the '>>' prompt in Matlab's 'Command window'.

‘VECTORS’ OR ‘ARRAYS’ IN MATLAB

Creating Our First Array

The most common way of storing data in Matlab is referred to as a ‘vector’, or an ‘array’. You can think of this as a list of numbers that your program can run through, either one at a time, or in chunks. For instance, if we want to create a list of numbers from 3 to 7, we would write:

```
> myVector = [3, 4, 5, 6, 7]

myVector =

    3    4    5    6    7
```

It is as easy as that! Now, we have a variable called ‘*myVector*’, which is a list of numbers from 3 to 7. Note that the command we need to type in is shown next to the ‘>>’ prompt, and the output from Matlab is shown below it.

Typing in these five numbers manually is no big deal, but what if you want to create a list of numbers that contains a hundred numbers, or even a million numbers? You would not enter each of the numbers manually! There is an easy and intuitive way of doing this automatically in Matlab. For instance, we could have created ‘*myVector*’ using the following command:

```
>> myVector = [3:7]

myVector =

    3    4    5    6    7
```

You should read the colon (:) as meaning ‘to’, i.e., the right hand side of the expression should be read as ‘3 to 7’. This gives us an identical list as the command

`myVector = [3, 4, 5, 6, 7]`.

Now, we can create vectors that are hundreds of elements long. For instance:

```
>> mySecondVector = [4:300];
```

will give you a list of numbers that starts at 4 and ends at 300, and contains all the integers 5, 6, 7, …, 299 in between. The semi-colon at the end of the statement just suppresses output from Matlab to our screen.

Note that the numbers stored in the vector do not necessarily have to be integers, but we will stick to integers for now for the sake of simplicity. Now that we know how to create a list or 'vector' of numbers, let us take a look at how we can manipulate this list by extracting and manipulating data.

Accessing vector 'elements'

Accessing single elements

An 'element' is one particular number stored in the vector. The location of this element in the vector is called the 'index' of that particular element. We can do several things with the vector we created in the previous section. For instance, we can look at the first element (or in other words, the element at index location 1):

```
>> myVector(1)

ans =

    3
```

We can look at the third element

```
>> myVector(3)

ans =

    5
```

or at the last element

```
>> myVector(end)

ans =

    7
```

Note that '*end*' is a Matlab keyword, which automatically gives you access to the very last element stored in the array. We could also have extracted the last element using the keyword '*length*' of the vector as follows:

```
>> myVector(length(myVector))

ans =

    7
```

Acessing multiple elements simultaneously

Now, what if we want to access 'chunks' of the vector, instead of just single elements? We can do this easily using the same 'to' construct we learned earlier, i.e., the colon (:).

```
>> myVector(1:3)

 ans =

    3    4    5
```

gives us back the first three (first 'to' third) elements of the vector,

```
>> myVector(2:4)

ans =

    4    5    6
```

gives us the elements starting at position 2, and ending at position 4 (2nd 'to' 4th element),

```
>> myVector(2:end)

ans =

    4    5    6    7
```

```
>> myVector(2:length(myVector))

ans =

    4    5    6    7
```

gives use the chunk of elements from position 2 'to' the end, and so on. You get the idea! This sort of operation is referred to as 'slicing', because you are effectively slicing up the array into chunks that are useful to you.

Striding

Now, suppose that the vector that you created contains useful information only on every other index, i.e., the useful data is located at index locations 1, 3, 5, … The simplest way to 'slice' this useful chunk out of the vector is as follows:

```
>> myVector(1:2:end)

ans =

    3    5    7
```

The argument (the statement '*1:2:end*' inside the parentheses) should be read as "start at index location 1, keep incrementing the index by 2, stop at the last index location".

This is referred to as 'striding', since we use a 'stride' of 2 to jump over some of the elements. Thus, using a stride of 1 (i.e., '*1:1:end*') would be the same as saying 'myVector(1:end)', i.e., it would return a chunk containing all the vector elements.

You can go ahead and try arguments like '*1:4:end*', and any other combinations you want, to get a feel for how striding works. Note that if your 'stride' tries to take you beyond the last element location in the vector, the last element will be skipped in the chunk you extract. For instance, try the following and see what happens

```
>> myVector(1:3:end)

ans =

    3     6
```

If you want to access the elements of the array in reverse order, the simplest way to do so is

```
>> myVector(end:-1:1)

ans =

    7     6     5     4     3
```

This time, we are starting at the *'end'*, striding by -1 (or in other words, moving backwards along the vector), and stopping at 1.

Accessing elements using explicit index locations

Now suppose that we know exactly which locations in the array we want to extract your data from. We can use an explicit list of array indices to slice our chunk out of the array:

```
>> myVector([2, 3, 5])

ans =

    4     5     7
```

This returns a chunk containing the elements at index location 2, 3, and 5 in *myVector*.

You might have recognized by now that striding does effectively the same thing as specifying an explicit list of array indices. When we say 'myVector(1:3:end)', Matlab automatically creates a new temporary vector that is hidden from us:

1:3:end = [1, 4]

(*Note that 7, 10, 13, … are not included here, because Matlab knows that the length of myVector is 5)*

and then returns the answer of the following to us

myVector([1,4])

Summary

That should be all you need to know for the time being to start playing around with vectors. If you think of some cool ways of doing these operations, go ahead and try them! This is the best way of learning what works and what does not. Besides, you might be surprised at how often your intuitive guess might be correct when writing simple code. Later on, we will learn how to do much more with vectors, such as inserting new elements, or deleting existing ones, performing several basic mathematical operations, etc.

Use in General Programming

'Loops' constitute one of the very basic building blocks of ***any*** programming language. So what exactly do loops do? They allow us to run through the list of numbers you stored in your vector, one by one. This is referred to as 'iterating' in programming lingo.

There are two main flavors of looping that you will encounter, the '*for*' loop, and the '*while*' loop. Almost any iterative task you want to perform can be accomplished using a '*for*' loop. I would advise you to avoid using '*while*' loops until you become a little more familiar with the basics of programming. If you are eager to learn the syntax, a quick web-search should give you the answer. However, keep in mind that you will almost never need to use while-loops; I use them very rarely in my work.

'*If*' conditionals are another very important part of programming. They allow us to specify chunks of code that the computer will execute if and only if certain conditions are true.

For-Loops

Let's reuse our vector to demonstrate how for-loops work. Recall that

```
>> myVector = [3:7]

myVector =

    3    4    5    6    7
```

If we want to visit each element in the vector, and output it to our screen, we would use the following syntax in Matlab:

```
>> for index = 1:length(myVector)
        myVector(index)
    end
```

In plain-speak, this is just telling the computer the following: one by one, set the value of a new variable '*index*' to be the numbers in the range '1:length(*myVector*)' (i.e., [1, 2, 3, 4, 5]), and print out the element from '*myVector*' at the corresponding location.

Being able to translate quickly this in your head is crucial.

So let us break down the loop into individual steps to see what exactly is going on. When the computer first sees the line

```
for index = 1:length(myVector)
```

It understands that it must get ready to do some sort of iterative task. It creates a new variable called '*index*' (we could have called this by any other name), and sets its value = 1. Then it proceeds to the next line

```
myVector(index)
```

Here, the computer just extracts myVector(1) (since the value of index is currently set equal to 1), and outputs it to our screen (ans = 3). With the instructions on the current line executed, the computer proceeds to the next line

```
end
```

This keyword tells the computer that the chunk of code that we wish to execute iteratively (i.e., repeat a specified number of times) ends at this line. At this point, the computer will return to the start of the iterative chunk

```
for index = 1:length(myVector)
```

The next value for *index* is at this point equal to 2. The computer then proceeds to the next line of the code, which again is

```
myVector(index)
```

This time around, the value of *index* is 2, so the computer executes *myVector(2)*, and outputs *ans = 4*. Then it hits the next line

```
end
```

At this point, the computer returns to the first line of the iterative chunk for the third time, knowing that it must execute the third loop

```
for index = 1:length(myVector)
```

And then goes on to output *myVector(3)*, and so on… You probably get the idea by now!

The chunk within the 'for-end' lines will keep repeating until the computer knows it is time to move on, i.e. until the computer sets the last value for *index*. For our specific case, this happens after the fifth iteration, i.e. after the computer has executed the iterative chunk with *index* = 5. Following this, the computer moves to the line that comes after 'end' and goes on its merry way!

'If' Conditionals

The if-statement is self-explanatory. You use it when you want a chunk of code to execute, only if a certain condition (specified by you) is true. Let us look at a simple example

```
> for index = 1:length(myVector)
                    if (myVector(index) > 4)
                            currentIndex = index
                            currentValue =   myVector(index)
                    end
        end
```

As before, the loop will run through all the elements in '*myVector*'. However, when the computer hits the line

```
if (myVector(index) > 4)
```

it will evaluate the expression contained within the parentheses. The computer will execute the code chunk contained within the 'if-end' block, if and only if this expression is true, i.e., if the current value of *myVector*(index) is greater than 4. Therefore, the output on your screen should look like

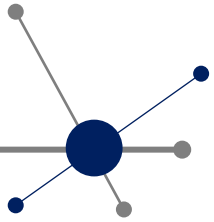
```
currentIndex =

    3

currentValue =

    5

currentIndex =

    4

currentValue =

    6

currentIndex =

    5

currentValue =

    7
```

Keep in mind that the code is actually checking the *'if (myVector(index) > 4)'* condition for all the elements in *myVector.* It will react with an output if and only if the particular element being inspected during the loops satisfies the condition we impose.

Summary

At this point, we have not even begun to scratch the surface of the wonderful things we can do with for-loops and if-statements. However, remember that our goal at this moment is to get comfortable with the basics. You might already be beginning to appreciate how for-loops and if-statements can be combined to accomplish virtually any task!

2D ARRAYS



Straightforward extensions of 1D arrays

2-Dimensional (2D) arrays are very simple extensions of the 1D arrays or vectors that we have already seen. If you remember using matrices at some point in your mathematics class, you might already be familiar with the basic concepts of how data can be stored and manipulated using 2D arrays.

One of the most important uses of 2D arrays is for storing time-series data. Of course, these arrays have innumerable other uses, and there are multiple ways you could store time-series data (in 'structures', for instance). However, 2D arrays, or matrices, offer the simplest way of storing and manipulating such data.

A Practical Example

Imagine that you are recording the vital signs of an athlete while he or she is exercising. The data you collect might look as follows:

Time [minutes]	Pulse [beats/min]	Respiratory Rate [breaths/min]	Temperature
0	64	10	36.8
2	78	15	37.0
5	120	20	37.5
8	130	30	38.0
10	128	28	37.9

Here, each row represents an independent time observation, and the columns represent different variables being observed. If we were to create a collection of this dataset, we would no longer just have a 'list' of numbers. Instead, we would have a 2-Dimensional array of numbers, and hence the name!

In Matlab, the simplest way to enter this data manually is as follows:

```
>> VitalSigns = [0, 64, 10, 36.8;
2, 78, 15, 37.0;
5, 120, 20, 37.5;
8, 130, 30, 38.0;
10, 128, 28, 37.9]
```

```
VitalSigns =

    0.0000   64.0000   10.0000   36.8000
    2.0000   78.0000   15.0000   37.0000
    5.0000  120.0000   20.0000   37.5000
    8.0000  130.0000   30.0000   38.0000
   10.0000  128.0000   28.0000   37.9000
```

The elements in the first row – 0, 64, 10, and 36.8 – are separated by commas. Then a semicolon (;) tells Matlab that the data to follow should go on the next row. You could even have entered this data using a single line, but then you run the risk of reducing ease of readability:

```
>> VitalSigns = [0, 64, 10, 36.8; 2, 78, 15, 37.0; 5, 120, 20, 37.5;  8, 130, 30, 38.0;  10, 128, 28, 37.9];
```

Obviously, this is not an optimal way of loading data into Matlab! You want to be able to read data automatically from a text file that your instruments might generate for you. We will learn how to do this in one of the later sections.

Accessing/extracting elements from 2D arrays

Accessing and extracting data from 2D matrices is similar to what we have already learned for 1D arrays/vectors. Suppose that we want to check out the element in the second row, fourth column of the matrix. The syntax to access this element is

```
>> VitalSigns(2, 4)
```

```
ans =

    37
```

If we want to access the element in the fifth column, third row, we would type

```
>> VitalSigns(5, 3)

ans =

   28
```

Notice that the first argument (e.g., the '5' in *VitalSigns(5,3)*) always refers to the row number we want to look at and the second argument refers to the column number. This syntax is the same as doing matrix mathematics on paper. If you did not already know, the name Matlab derives from 'Matrix Laboratory', which means that the language is singularly suited to making large and complicated matrix calculations easy for you!

Slicing, Striding and Explicit Indexing in 2D Arrays

Now, let us start 'slicing' out data from our 2D array. Suppose that we want to list our athlete's respiratory rates from the table, i.e., we want to look at elements from all the rows in the 3rd column. To retrieve this data from our '*VitalSigns'* matrix, we would type:

```
>> VitalSigns( :, 3)

ans =

   10
   15
   20
   30
   28
```

Notice that we have made use of the colon ('*:*' in the first argument) for extracting elements from '*all*' of the rows. The second argument indicates that the data we want resides only in the third column.

To extract only the first two rows in the third column, we would have used

```
>> VitalSigns( 1:2, 3)

ans =

   10
   15
```

Suppose that you want to examine the athlete's breathing-rate as a function of time (in later sections, we will learn how to make simple plots to examine such data). To get the

relevant data, we would need to extract elements in all the rows, but only in the first and third column:

```
>> VitalSigns( :, [1, 3])

ans =

    0    10
    2    15
    5    20
    8    30
   10    28
```

If we want to examine Time and Breathing-rate data only from every other row, we could have used:

```
>> VitalSigns( 1:2:end, [1, 3])

ans =

    0   10
    5   20
   10   28
```

Once again, we have used 'striding' (*1:2:end*) to skip over the second and the fourth rows, just as in the case of 1D arrays.

If we want to look at the data in reverse order:

```
>> VitalSigns( end:-1:1, [1, 3])

ans =

   10   28
    8   30
    5   20
    2   15
    0   10
```

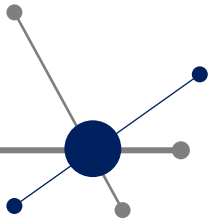
Keep in mind that you are not restricted to using these operations (such as slicing, striding, or listing in reverse order) on a specific dimension of the matrix. In other words, you could just as easily have applied these operations to skip over the columns, or reverse them, instead of the rows.

Knowing how to perform such basic operations is absolutely essential for us to be able to do much more useful things in Matlab. So before moving on, make sure that you get comfortable with the idea of accessing data chunks stored in matrices.

Summary

Now that we are familiar with both 1D and 2D arrays, you must understand that there is nothing special about the 'dimensionality' of an array. Matlab can easily handle n-Dimensional arrays; where 'n' can be any arbitrary number of your choice. The pattern for accessing data elements from such arrays remains the same as we have already seen for 1 and 2 dimensions. However, for most of your data analysis, you will very rarely need to use dimensions higher than 2.

IMPORTING DATA



Loading Data from Text Files

Generating the matrix containing the athlete's data was relatively simple to do manually. How will we manage when we encounter real-world data with thousands of numbers to be stored in the matrix? Usually, such data is stored as comma- or tab-separated values in a text file. If your data is stored in an Excel spreadsheet, you can easily export to comma-separated (CSV) files using the 'save as' option. There are built-in commands in Matlab, which can painlessly import this data for you.

Suppose that you have a text file called '*athlete1.txt*', which contains data separated by commas:

```
0,64,10,36.8
2,78,15,37
5,120,20,37.5
8,130,30,38
10,128,28,37.9
```

Note that this dataset is identical to the one we used when getting familiar with 2D arrays/matrices. The cleanest way of importing these numbers into Matlab is using the '*importdata*' command:

```
>> Athlete1 = importdata('athlete1.txt')

Athlete1 =

    0.0000   64.0000   10.0000   36.8000
    2.0000   78.0000   15.0000   37.0000
    5.0000  120.0000   20.0000   37.5000
    8.0000  130.0000   30.0000   38.0000
   10.0000  128.0000   28.0000   37.9000
```

There are other alternatives like '*csvread*', or '*dlmread*' for importing data, but '*importdata*' will almost always be the least troublesome to use.

Text files with headers

If your text files have headers (i.e., text indicating what variable each column contains), '*importdata*' is smart enough to recognize this. However, we need an additional line to retrieve the 2D matrix containing the data. Suppose that our 'athlete1.txt' text file, including the header, looks as follows:

```
Time,Pulse,BreathRate,Temp
0,64,10,36.8
2,78,15,37
5,120,20,37.5
8,130,30,38
10,128,28,37.9
```

With this file, '*importdata*' will return the following output, which we are not familiar with at present

```
>> Athlete1 = importdata('athlete1.txt')

Athlete1 =

data: [5x4 double]
textdata: {'Time'  'Pulse'  'BreathRate'  'Temp'}
colheaders: {'Time'  'Pulse'  'BreathRate'  'Temp'}
```

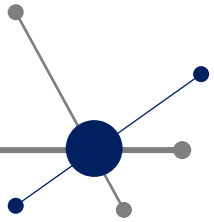
To extract the numerical data, all we need to do is the following:

```
>> VitalSigns = Athlete1.data

VitalSigns =

  0.0000   64.0000   10.0000   36.8000
  2.0000   78.0000   15.0000   37.0000
  5.0000  120.0000   20.0000   37.5000
  8.0000  130.0000   30.0000   38.0000
 10.0000  128.0000   28.0000   37.9000
```

We can start analyzing this data whichever way we please. It is easy to import real-world data that may contains hundreds of thousands of numbers, with very few hassle-free commands.

Just in case you are curious, when the file being read has header text. 'importdata' generates a 'structure' instead of a 2D matrix.
'Structures' are slightly advanced, although extremely useful tools for grouping together data in a logical way, which we will not discuss in this book.

MATHEMATICAL OPERATIONS



Simple Operations

Doing simple mathematics in Matlab is quite straightforward. If you want to add/ subtract/divide/multiply a number to a vector, you can simply type in what you would intuitively expect to work!

Let us extract our athlete's core temperature from the data given to us:

```
>> coreTemperature = VitalSigns(:,end)

coreTemperature =

   36.8000
   37.0000
   37.5000
   38.0000
   37.9000
```

Now, let us convert these values from degrees Centigrade to Fahrenheit (the formula we need to use is C = (5/9)*(F-32)). We first need to multiply the Centigrade values by (9/5), and then add 32:

```
>> fahrenheitTemp = (coreTemperature*9/5) + 32

fahrenheitTemp =

   98.2400
   98.6000
   99.5000
  100.4000
  100.2200
```

Note that for these operations, we have added or multiplied a single number to ***all*** the numbers stored in our vector.

There are times when we will need to do element-by-element mathematical operations as explained in the following section.

Element-wise operations

Element-wise mathematics is useful when we have two or more arrays (1D, 2D, or n-D) of the same shape and size, and we want to perform a collective operation on the two.

As an example, suppose that we want to perform an element-wise sum on two matrices. For this operation, the element at location (1,1) in matrix A would be added to the element at location (1,1) in matrix B; the element at location (1,2) in A would be added to the element at location (1,2) in B, and so on. These operations should become clear from the following examples:

```
>> A = [2, 4; 3, 7]

A =

    2    4
    3    7

>> B = [5, 6; 8, 4]

B =

    5    6
    8    4
```

For element-wise sum and difference, we can simply type in:

```
>> A + B

ans =

    7   10
   11   11

>> A - B

ans =

   -3   -2
   -5    3
```

Element-wise multiplication and division is slightly different, and requires that we use '.*' instead of '*', and './' instead of '/'.

```
>> A.*B

ans =

   10   24
   24   28

>> A./B

ans =

    0.4000    0.6667
    0.3750    1.7500
```

Just in case you are curious, using the '*' and '/' operators on matrices results in 'matrix-multiplication' and 'matrix-division'. Do not worry if you are not familiar with this concept, you will rarely ever need it for basic data processing.

If you are familiar with matrix algebra, A*B and A/B (which is the equivalent of 'A*inverse(B)') give the results you would expect:

```
>> A*B

ans =

   42   28
   71   46

>> A/B

ans =

    0.8571   -0.2857
    1.5714   -0.6071
```

Vector Operations

There are several useful built-in mathematical functions (commands), to which you can feed in entire vectors. Most of these commands will work with single numbers (scalars) as well.

We will generate a new vector to demonstrate some simple ways to use these commands:

```
>> realVector = [-5, -8, 100, 45, 0.8, -0.1]

realVector =

   -5.0000   -8.0000  100.0000   45.0000    0.8000   -0.1000
```

If you want to determine the minimum or the maximum value stored in this vector, type in:

```
>> min(realVector)

ans =

    -8

>> max(realVector)

ans =

   100
```

If you want to figure out where exactly the min/max values are located, use the following command

```
>> [minValue, minLocation] = min(realVector)

minValue =

    -8

minLocation =

     2
```

The second argument '*minLocation*' returns the index location of the smallest element in the vector, and the first argument '*minValue*' returns the smallest element itself.

Now, suppose that we want to add up all the elements in the vector, or multiply them all together. We could do this using:

```
>> vectorSum = sum(realVector)

vectorSum =

  132.7000

>> vectorProduct = prod(realVector)

vectorProduct =

      -14400
```

If we want to return the absolute values of all the elements in the vector, we would type in:

```
>> abs(realVector)

ans =

    5.0000    8.0000  100.0000   45.0000    0.8000    0.1000
```

We could raise all the elements in the vector to whatever power we want using the '.^' operator. Again, note the use of '.^' for element-wise operations, instead of '^':

```
>> realVector.^2

ans =

   1.0e+04 *

    0.0025    0.0064    1.0000    0.2025    0.0001    0.0000
```

Suppose that you want to compute some statistical quantities for the data stored in a vector, like its mean, standard deviation, or the variance. Let us use the 'fahrenheit' vector from one of the previous sections:

```
>> fahrenheitTemp = (coreTemperature*9/5) + 32

fahrenheitTemp =

   98.2400
   98.6000
   99.5000
  100.4000
  100.2200

>> meanVal = mean(fahrenheitTemp)
standardDev = std(fahrenheitTemp)
variance = var(fahrenheitTemp)

meanVal =

   99.3920

standardDev =

    0.9576

variance =

    0.9169
```

You could even use the '*round*', '*floor*', or '*ceil*' (ceiling) commands to convert decimal numbers into appropriate integers:

```
>> round(fahrenheitTemp)

ans =

    98
    99
   100
   100
   100

>> floor(fahrenheitTemp)

ans =

    98
    98
    99
   100
   100

>> ceil(fahrenheitTemp)

ans =

    99
    99
   100
   101
   101
```

We could take the logarithm or the square root of all the elements in a vector as follows:

```
>> vectorLogarithm = log(fahrenheitTemp)
vectorSquareRoot = sqrt(fahrenheitTemp)

vectorLogarithm =

    4.5874
    4.5911
    4.6002
    4.6092
    4.6074

vectorSquareRoot =

    9.9116
    9.9298
    9.9750
   10.0200
   10.0110
```

Logical Operators

Just like for-loops and if-conditionals, logical operators are common to almost all programming languages. These operators can be used to determine:

- whether two or more specified conditions are all true simultaneously
- whether at least one of the specified conditions is true
- or a number of other variations

The three operators you will need at the moment are

‘AND’ - &&
‘OR’ - ||
‘NOT’ - ~

The following examples should help clarify how these operators work.

```
>> a = 4;
b = -8;
(a > 0) && (b>0)

ans =

    0
```

In plain-speak, the third line of the command '(a > 0) && (b>0)' is asking Matlab to check whether '*a*' is greater than zero AND at the same time '*b*' is greater than zero. Matlab returns a '0', telling us that the answer to our question is 'false'. Similarly:

```
>> (a > 0) && (b<0)

ans =

    1
```

returns '1' for 'true'.

For the 'AND' operator to return 'true', all the conditions that are 'chained together' must be true. However, when using the 'OR' operator, only one of the conditions need to be valid to return 'true':

```
>> (a > 0) || (b>0)

ans =

    1
```

This statement returns 'true' because the first condition ('a' is greater than 0) is true. It does not matter that the second condition ('b' is greater than 0) is false.

The ‘NOT’ operator returns the opposite of the answer determined by the argument:

```
>> ~(a>0)

ans =

    0
```

returns a ‘false’ because the argument (a>0) is true. Similarly, if the argument were false, the ‘NOT’ operator would return ‘true’:

```
>> ~(b>0)

ans =

    0
```

Slightly advanced use of logical operators:

Be careful when chaining commands, and make use of parentheses as much as possible. For instance, it would be difficult to predict the outcome of the following statement at first glance:

```
>> a = 4;
b = -8;
c = 0.5;
d = -98;
(a > 0) && (c>=0.2) || (b>0) && (d>100)
```

Although you might have guessed the answer correctly, statements that are more complicated might trip you up later on. In this statement, it is difficult to predict whether the ‘&&’ in front of (d>100) will cause the entire statement to be false, or whether Matlab will be content with the fact that the chunk (a > 0) && (c>=0.2) in front of the ‘||’ operator is true, and return ‘true.’ We should modify this expression using parentheses to make our intent explicitly clear..

Just like in regular algebra, you can get very different results depending on where exactly you put the parentheses:

```
>> ((a > 0) && (d>100) && (c>=0.2)) || (b<0)

ans =

     1
```

In this statement, we have separated our chain into two chunks, one before the '||' operator and one after.

Matlab checks both these chunks, and returns 'true', because the second chunk is true. The validity of the first chunk does not matter because of the 'OR' operator.

The first chunk '((a > 0) && (d>100) && (c>=0.2))' is false because not ALL of the conditions specified here are true: '*d*' is not greater than 100.

Moving the parentheses to a different place can give you the exact opposite result:

```
>> ((a > 0) && (d>100)) && ((c>=0.2) || (b<0))

ans =

     0
```

Now, we have separated the statement into two chunks: '((a > 0) && (d>100))' before the '&&' operator, and '((c>=0.2) || (b<0))' after it.

The first chunk is false again because '*d*' is not greater than 100.

However, the second chunk is true, because b is negative, as well as '*c*' is greater than or equal to 0.2. Note that only one of these *need* to be true for the second chunk to return true, because of the 'OR' operator in there.

The final result however is 'false', because the '&&' operator in between the first and second chunk requires that both the chunks be valid, which is not the case!

Do not worry if you don't get expected results at first. With a little tinkering, you will get accustomed to using these operators correctly.

We are now at a point where the essential basic commands in Matlab are familiar to us. We will now go through short descriptions of several commands that we can use for handling and manipulating data. Several of these built-in commands do exactly what you would expect them to do intuitively!

Let us first create a new 1D array that we can play around with, we have already seen how to use ':' to generate a list of integers

```
>> integerVector = 5:3:12

integerVector =

    5     8    11
```

We start at 5, increment with a stride of 3 (i.e., the elements are 5, (5)+3, (5+3)+3), and end at 12. Matlab stops with 11 as the last element, since using a stride of 3 past 11 (11 + 3) would take us past the specified last element, which is 12.

Doing the same for real numbers (i.e., decimal numbers) involves exactly the same syntax.

```
>> realNumbers = 0:0.2:0.9

realNumbers =

         0    0.2000    0.4000    0.6000    0.8000
```

We could have typed this vector out manually, since it contains just a handful of elements. However, knowing how to generate vectors that are hundreds or thousands of elements long is essential when working with real-world data.

'Size': for Determining the Size of an Array

This built-in command returns the size of a matrix, i.e., the number of rows and columns that the matrix contains. We can use it on our '*VitalSigns*' matrix as follows:

```
>> size(VitalSigns)

ans =

    5     4
```

The output tells us that the matrix '*VitalSigns*' contains 5 rows and 4 columns. Remember that whenever talking about 2D matrices, the first index always corresponds to rows, and the second one to columns, just as in matrix algebra.

The '*size*' command is useful when you want to figure out how many observations (rows) or variables (columns) a particular dataset might hold. We also use the command quite frequently when iterating over the rows or columns of a matrix (2D array) using for-loops.

'Ones' and 'zeros': for Initializing Arrays

The commands '*ones'* and '*zeros*' create arrays of specified size, with all the elements of the array set to 1 or 0, respectively:

```
>> ones(3,5)

ans=

1    1    1    1    1

1    1    1    1    1

1    1    1    1    1

>> zeros(4,2)

ans =

     0     0

     0     0

     0     0

     0     0
```

Arrays created in this manner are quite useful when we want to accumulate results by either summing up numbers (in which case you would initialize with '*zeros*'), or by multiplying them (initialize with '*ones*').

'Linspace': for Generating Uniformly Spaced Numbers

Very often, you will need to work with uniformly spaced numbers. For example, you might want to create a vector of the time instances when your camera captures a frame, when it operates in continuous-shoot (or 'burst') mode. You know that your camera shoots 5 images per second in this mode. To generate the appropriate time-stamps, we use:

```
>> frameTimes = linspace(0, 1, 5)

frameTimes =

     0    0.2500    0.5000    0.7500    1.0000
```

This command is instructing Matlab to start at 0 (the first argument), stop at 1 (the second argument), and create 5 (the third argument) uniformly spaced points in between, including the two numbers.

Note that the third argument specifies the number of points, and not the number of divisions: the number of divisions is always 1 less than the number of points. Thus, with '*linspace(0, 1, 5)*', we are instructing Matlab to divide the space from 0 to 1 seconds into (5-1) = 4 divisions. Each division will thus measure 0.25 seconds, as we can clearly see from the result '*frameTimes*'.

Similarly, linspace(0,1,4) would give us [0, 0.3333, 0.6667, 1.0000], and not [0, 0.25, 0.5, 0.75, 1.0]. Even now, when I am coding in a hurry, I sometimes make the mistake of expecting the second result!

'Plot': for plotting data

Imagine that you want to plot a sine wave over the range 0 to 4*pi. For this, we first need to generate discrete data points along the x-axis. Let us say that we want to use a total of 100 points

```
>> theta = linspace(0, 4*pi, 100);
```

(Note: 'pi' is a built-in constant in Matlab)

We can easily compute the corresponding y-values

```
>> y = sin(theta);
```

And create the plot

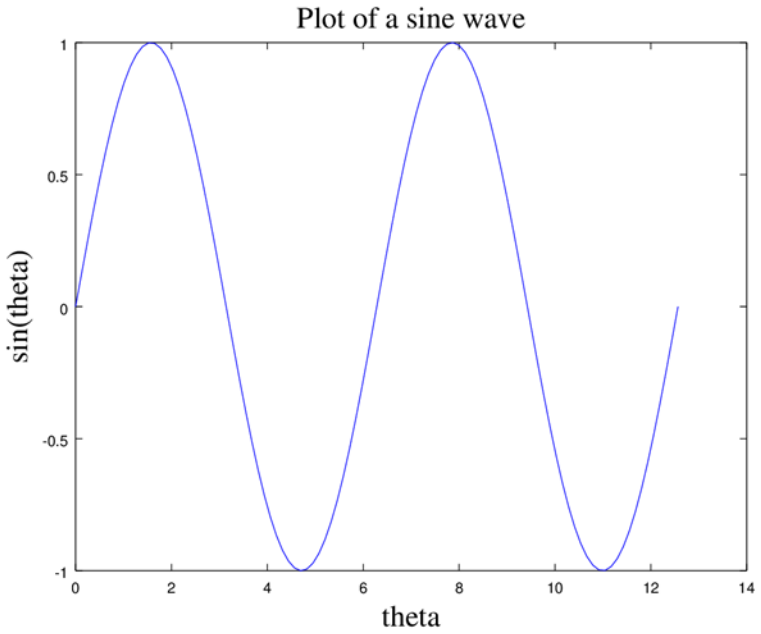
```
>> plot(theta, y)
```

The first argument is assigned to the x-axis (abscissa), and the second argument is assigned to the y-axis (ordinate). That is it! We are done with the plotting, all with a single command!

We can add some decoration, such as labeling the axes and adding a title

```
>>xlabel('theta')
ylabel('sin(theta)')
title('Plot of a sine wave')
```

The resulting plot looks as follows



There are fantastic things you can do with the 'plot' command! Just head over to the mathworks website to see for yourself. We will discuss most of the advanced plotting techniques in a separate volume.

'Find': for finding data

The '*find*' command returns the index locations of the elements that satisfy the specified condition. Let us use the vector '*realNumbers*' to see how this works.

```
>> realNumbers = 0:0.2:0.9

realNumbers =

         0    0.2000    0.4000    0.6000    0.8000

>> find(realNumbers >= 0.4)

ans =

    3     4     5

>> find(realNumbers > 1)

ans =

   Empty matrix: 1-by-0
```

```
>> find((realNumbers+0.3) > 1)

ans =

     5
```

If we want to extract the contents of '*realNumbers*' that satisfy the specified condition, we can store the index locations that '*find*' returns, and extract the corresponding elements as follows:

```
>> indexLocations = find(realNumbers > 0.5)

indexLocations =

     4     5

>> realNumbers(indexLocations)

ans =

    0.6000    0.8000
```

'Sort': for sorting data

Sorting data in ascending or descending order is a common task that we might need to perform. Matlab's built-in '*sort*' command effortlessly does the job for us:

```
>> unsortedData = [100, -4, 6, 87, -88, 45]

unsortedData =

   100    -4     6    87   -88    45

>> sortedData = sort(unsortedData)

sortedData =

   -88    -4     6    45    87   100
```

One very useful feature of the '*sort*' command that I use quite frequently is its ability to return the sorting index locations:

```
>> [sortedData, sortingIndex] = sort(unsortedData)

sortedData =

   -88    -4     6    45    87   100

sortingIndex =

     5     2     3     6     4     1
```

This comes in very handy when you want to sort several different variables according to the ascending/descending order of one particular variable (for example, if you want to arrange variables according to increasing time). We will learn how to do this in the example below. Let us look at the vital signs of our athlete again.

```
>> VitalSigns = [0, 64, 10, 36.8;
2, 78, 15, 37.0;
5, 120, 20, 37.5;
8, 130, 30, 38.0;
10, 128, 28, 37.9]
VitalSigns =

    0.0000   64.0000   10.0000   36.8000
    2.0000   78.0000   15.0000   37.0000
    5.0000  120.0000   20.0000   37.5000
    8.0000  130.0000   30.0000   38.0000
   10.0000  128.0000   28.0000   37.9000
```

Recall that the first column is time, the second column is pulse, the third column is breathing rate, and the last column is the core temperature. Suppose our job is to arrange the data in order of increasing pulse.

First, we must determine what the correct ascending order for the '*pulse*' column (i.e., column number 2) is:

```
>> pulse = (VitalSigns(:,2))

pulse =

    64
    78
   120
   130
   128

>> [sortedPulse, sortingIndex] = sort(pulse)

sortedPulse =

    64
    78
   120
   128
   130

sortingIndex =

     1
     2
     3
     5
     4
```

Now, we must rearrange the rows (or in other words, the observations) in our '*VitalSigns*' matrix, such that all of the data corresponds to increasing pulse:

```
>> VitalSigns(sortingIndex,:)

ans =

            0   64.0000   10.0000   36.8000
    2.0000   78.0000   15.0000   37.0000
    5.0000  120.0000   20.0000   37.5000
   10.0000  128.0000   28.0000   37.9000
    8.0000  130.0000   30.0000   38.0000
```

We have managed to arrange the data in order of increasing pulse! The resulting matrix is similar to the original '*VitalSigns*' matrix, but with the last two rows swapped. Do not worry if it takes you a few tries to get used to such operations!

To break down what we did:

1) First, we found the correct ordering for the rows, assuming that the data is to be arranged in order of increasing pulse. This was done using the '*sort*' command, and saving the indices it returns as '*sortingIndex*'. The numbers output in '*sortingIndex*' tell us that row 1 comes first then row 2, then row 3, then row 5, and finally row 4.

2) Next, we examined what the output with the correctly ordered rows would look like. For this, we used explicit indexing (as explained in detail in the section titled: Slicing, striding and explicit indexing in 2D arrays on the 2D array '*VitalSigns*'. Remember that we do not want to change the ordering of the columns, since we still want the first column of the output to be time, the second column to be the pulse, and so on. This is the reason why we use the colon operator (:) as the second argument in '*VitalSigns(sortingIndex,:)*'. However, the rows (which represent observations in our case) must be arranged according to the values in '*sortingIndex*', which is what the first argument in the command achieves.

We could just as easily have arranged the data in the order of decreasing pulse. To do this, we can simply use our '*sortingIndex*' vector in reverse order:

```
>> VitalSigns(sortingIndex(end:-1:1),:)

ans =

    8.0000  130.0000   30.0000   38.0000
   10.0000  128.0000   28.0000   37.9000
    5.0000  120.0000   20.0000   37.5000
    2.0000   78.0000   15.0000   37.0000
         0   64.0000   10.0000   36.8000
```

Another alternative is to first arrange the matrix in order of increasing pulse, and then use reverse indexing on the rows of this temporary matrix:

```
>> tempMatrix = VitalSigns(sortingIndex,:)
reversedMatrix = tempMatrix(end:-1:1,:)
tempMatrix =
         0   64.0000   10.0000   36.8000
    2.0000   78.0000   15.0000   37.0000
    5.0000  120.0000   20.0000   37.5000
   10.0000  128.0000   28.0000   37.9000
    8.0000  130.0000   30.0000   38.0000
reversedMatrix =
    8.0000  130.0000   30.0000   38.0000
   10.0000  128.0000   28.0000   37.9000
    5.0000  120.0000   20.0000   37.5000
    2.0000   78.0000   15.0000   37.0000
         0   64.0000   10.0000   36.8000
```

Matrix transpose

A matrix transpose in Matlab is the same as its mathematical counterpart: columns become rows, and rows become columns. The single quote (‘) is used in Matlab as the transpose operator.

We can change a row vector into a column vector:

```
>> rowOfOnes = ones(1,4)
rowOfOnes'
rowOfOnes =
     1     1     1     1
ans =
     1
     1
     1
     1
```

We can change column vectors into row vectors:

```
>> columnOfZeros = zeros(3,1)
columnOfZeros'
columnOfZeros =
     0
     0
     0
ans =
     0     0     0
```

Or we can swap the rows and columns of a 2D matrix:

```
>> VitalSigns'
ans =
         0    2.0000    5.0000    8.0000   10.0000
   64.0000   78.0000  120.0000  130.0000  128.0000
   10.0000   15.0000   20.0000   30.0000   28.0000
   36.8000   37.0000   37.5000   38.0000   37.9000
```

Summary

By now, you are familiar with most of the basic commands that are the workhorses of programming in Matlab. You may combine these commands in various ways, to accomplish even the most challenging tasks. We will use our knowledge of just this collection of about 20 to 30 commands to solve a simple practical problem in the section Threading it all together.

MISCELLANEOUS IMPORTANT ITEMS

In this section, I will mention a few things that I could not fit-in nicely in any of the other sections, but which are important for you to know.

Matlab scripts

Up until now, we have been entering commands directly at the '>>' prompt in the Matlab command window. However, this is inconvenient for accomplishing most tasks, since we usually need to piece together a number of individual commands in a working program.

Matlab has an extremely useful built-in editor for this purpose. Just hit the button called '*New script*' in the 'Home' tab in Matlab's window, and we are ready to code! You can type out your instructions here, exactly as we did at the '>>' prompt, then save the program as a '.m' file. Running the code contained in m-files is as easy as hitting the 'Run' button in the 'Editor' tab.

I regularly start all my m-files with the following three commands, right at the very top:

```
clear
close all
clc
```

These commands ensure that any leftover variables from previous runs are cleared from Matlab's memory. Doing this for every single m-file you write might save you a lot of hassle trying to figure out why things might not be behaving the way you expect them to.

Manipulating matrices

We have already learned how to handle vectors and matrices in the previous sections. However, we never really considered the possibility of *changing* a vector or a matrix. We will now learn how to do this.

Changing existing elements

Let us reuse some of the 1D and 2D arrays that we have already worked with:

```
>> myVector = [3:7]

myVector =

    3    4    5    6    7
```

Suppose that we want to change the element situated at index location 4 (which in this case is '6') to -15. We can simply type in:

```
>> myVector(4) = -15

myVector =

    3    4    5   -15    7
```

If you want to change multiple elements, just use the appropriate indexing:

```
>> myVector(2:4) = [5.4, 7.2, 9.3]

myVector =

   3.0000    5.4000    7.2000    9.3000    7.0000

>> myVector([1, 5]) = [-12, -8.34]

myVector =

  -12.0000    5.4000    7.2000    9.3000   -8.3400
```

We can do the same for 2D matrices, with the main difference being that our index-location now consists of two numbers: the first number indicating the row, and the second indicating the column. For example, we can change the element in the 3rd row, 4th column of our 'VitalSigns' matrix to -140 as follows:

```
>> VitalSigns = [0, 64, 10, 36.8;

2, 78, 15, 37.0;

5, 120, 20, 37.5;

8, 130, 30, 38.0;

10, 128, 28, 37.9];

>> VitalSigns(3,4) = -140

VitalSigns =

         0   64.0000   10.0000   36.8000

    2.0000   78.0000   15.0000   37.0000

    5.0000  120.0000   20.0000 -140.0000

    8.0000  130.0000   30.0000   38.0000

   10.0000  128.0000   28.0000   37.9000
```

Concatenating data onto vectors

Instead of changing existing elements, you might want to increase the size of an array by adding-on more data. 'Concatenating' is the technical term used for the act of 'adding-on' stuff to an already existing vector or matrix.

Suppose that we want to insert a number at the beginning of an existing vector. We would use:

```
>> elementAtBeginning = [-500, myVector]

elementAtBeginning =

  -500     3     4     5     6     7
```

As you have probably already figured out, this statement simply says the following: "generate a new vector called 'elementAtBeginning', and populate it with a -500, followed by all the elements contained in '*myVector*' ".

Similarly, we could have used the following command to concatenate an element at the end of the vector:

```
>> elementAtEnd = [myVector, 38.76]

elementAtEnd =

3.0000    4.0000    5.0000    6.0000    7.0000   38.7600
```

If we want to insert a chunk (say, [-12 -11 -10 -9 -8 -7 -6]) somewhere in the middle of the vector, we would need to split the existing vector using slicing:

```
>> expandedVector = [myVector(1:3), -12:-6, myVector(4:end)]

expandedVector =

     3     4     5   -12   -11   -10    -9    -8    -7    -6     6     7
```

Notice that we have concatenated vectors of different sizes (*myVector* has 5 elements, -12:-6 has 7 elements) for creating *'expandedVector'*. This is fine, as long as you keep the number of rows the same, i.e, 1.

We can concatenate column vectors using ';' instead of ',' as our separator. We just have to make sure that the number of columns remains 1:

```
>> myVectorTranspose = myVector'

myVectorTranspose =

    3

    4

    5

    6

    7

>> expandedColumnVector = [myVectorTranspose(1:3); (-12:-10)'; myVectorTranspose(4:end)]

expandedColumnVector =

    3

    4

    5

  -12

  -11

  -10

    6

    7
```

Concatenating data onto matrices

Concatenating data onto matrices is not very different from concatenating data onto vectors. You just have to be careful that you increase only one thing at a time: either the number of rows, or the number of columns. Let us use our athlete's data again to see how this works:

```
>> VitalSigns = [0, 64, 10, 36.8;

2, 78, 15, 37.0;

5, 120, 20, 37.5;

8, 130, 30, 38.0;

10, 128, 28, 37.9];
```

Suppose that we want to add an extra variable onto our Athlete's data matrix: the amount of carbon dioxide (CO_2) he/she releases while exercising (measured as a percentage of total air volume exhaled):

```
>> CO2released = [4.2, 4.8, 5.2, 5.3, 5.5]

CO2released =

    4.2000    4.8000    5.2000    5.3000    5.5000
```

First, we have to convert this row vector into a column vector by taking the transpose, since we want to concatenate a column to our matrix:

```
>> CO2released = CO2released'

CO2released =

    4.2000

    4.8000

    5.2000

    5.3000

    5.5000
```

Alternatively, we could have defined '*CO2released*' as a column vector, by replacing the ',' separator with the ';' separator:

```
>> CO2released = [4.2; 4.8; 5.2; 5.3; 5.5]

CO2released =

    4.2000

    4.8000

    5.2000

    5.3000

    5.5000
```

Now, we can concatenate this column vector onto our matrix:

```
>> AdditionalVitalSigns = [VitalSigns, CO2released]

AdditionalVitalSigns =

         0   64.0000   10.0000   36.8000    4.2000
    2.0000   78.0000   15.0000   37.0000    4.8000
    5.0000  120.0000   20.0000   37.5000    5.2000
    8.0000  130.0000   30.0000   38.0000    5.3000
   10.0000  128.0000   28.0000   37.9000    5.5000
```

Beware that concatenation will work if and only if the number of rows is the same in both the matrices, when you concatenate columns (and if and only if the number of columns is the same, when concatenating rows). When this is not the case, you will get an error from Matlab saying "Error using horzcat/vertcat. Dimensions of matrices being concatenated are not consistent."

Now, suppose that you want to add an extra observation (i.e., a new row) at the end of the matrix. The new time, pulse, breathing rate, temperature, and CO_2 release data might look as follows:

```
>> newObservation = [12, 132, 26, 37.7, 5.4]

newObservation =

   12.0000  132.0000   26.0000   37.7000    5.4000
```

We can concatenate this row to the end of the matrix using:

```
>> AthleteExtraObservation = [AdditionalVitalSigns; newObservation]

AthleteExtraObservation =

         0   64.0000   10.0000   36.8000    4.2000
    2.0000   78.0000   15.0000   37.0000    4.8000
    5.0000  120.0000   20.0000   37.5000    5.2000
    8.0000  130.0000   30.0000   38.0000    5.3000
   10.0000  128.0000   28.0000   37.9000    5.5000
   12.0000  132.0000   26.0000   37.7000    5.4000
```

Note the use of the ';' separator instead of the ',' separator, since we are adding-on to the bottom of the matrix, not to the side.

Try the following command for yourself and see what happens.

```
>> AthleteExtraObservation = [VitalSigns; newObservation]
```

Think about how you might modify the '*newObservation*' vector (for instance by adding new elements to this vector, or by taking only some of the elements from this vector, etc.) to make this command work correctly, i.e., to get the following result

```
AthleteExtraObservation =

         0   64.0000   10.0000   36.8000
    2.0000   78.0000   15.0000   37.0000
    5.0000  120.0000   20.0000   37.5000
    8.0000  130.0000   30.0000   38.0000
   10.0000  128.0000   28.0000   37.9000
   12.0000  132.0000   26.0000   37.7000
```

Deleting data

Deleting data is not very different from concatenating. The operator used for deleting elements is '[]'. The following examples should clarify how this operator works.

```
>> expandedVector = [3, 4, 5, -12, -11, -10, -9, -8, -7, -6, 6, 7]

expandedVector =

     3     4     5   -12   -11   -10    -9    -8    -7    -6     6     7

>> expandedVector(4) = []

expandedVector =

     3     4     5   -11   -10    -9    -8    -7    -6     6     7
```

This command has removed the 4th element from '*expandedVector*', and consequently changed the size of the vector! If you do not want to touch the original vector, you must create a copy of this vector before the 'delete' operation:

```
>> expandedVector = [3, 4, 5, -12, -11, -10, -9, -8, -7, -6, 6, 7];

vectorCopy = expandedVector;

vectorCopy(4) = []

expandedVector

vectorCopy =

     3     4     5   -11   -10    -9    -8    -7    -6     6     7

expandedVector =

     3     4     5   -12   -11   -10    -9    -8    -7    -6     6     7
```

To remove multiple elements from the vector, we can use either slicing, or explicit indexing:

```
>> expandedVector = [3, 4, 5, -12, -11, -10, -9, -8, -7, -6, 6, 7]

expandedVector(3:6) = []

expandedVector =

    3    4   -9   -8   -7   -6    6    7

>> expandedVector = [3, 4, 5, -12, -11, -10, -9, -8, -7, -6, 6, 7]

expandedVector([4, 7, 9]) = []

expandedVector =

    3    4    5  -11  -10   -8   -6    6    7
```

Let's now look at how we can remove rows and columns from matrices. Suppose that our third observation during the athlete experiment was erroneous, and we want to delete it from the dataset. This means that we need to delete the entire 3rd row from the '*VitalSigns*' matrix:

```
>> VitalSigns(3,:) = []

VitalSigns

         0   64.0000   10.0000   36.8000
    2.0000   78.0000   15.0000   37.0000
    8.0000  130.0000   30.0000   38.0000
   10.0000  128.0000   28.0000   37.9000
```

The command (:) has deleted all columns in the 3rd row, as expected.

Now, you decide that you do not really need to keep track of all of the vital signs, but are interested only in the breathing rate of the athlete.

This means that we can delete the 2nd and 4th columns from our data matrix:

```
>> interestingData = VitalSigns;
interestingData(: , [2,4]) = []
interestingData =
    0    10
    2    15
    8    30
   10    28
```

We could just as easily have accomplished this by extracting the appropriate data, instead of deleting the unnecessary values:

```
>> interestingData = VitalSigns(:, [1,3])
interestingData =
    0    10
    2    15
    8    30
   10    28
```

Note that you can only remove either entire rows or entire columns from matrices. If you try to remove only a single element, the corresponding row and column would become inconsistent with the rest of the matrix, and Matlab angrily throws an error at you: *"Subscripted assignment dimension mismatch."*

Exporting Data

We learned about importing data from existing text files, in the section Importing Data. What about when we want to export data after performing some calculations? This too, is quite straightforward using commands already built into Matlab.

Let's take our athlete's data, change the core temperature values from degrees Centigrade to Fahrenheit, store the info in a new matrix, and write it out to a text file.

```
>> VitalSigns = [0, 64, 10, 36.8;
2, 78, 15, 37.0;
5, 120, 20, 37.5;
8, 130, 30, 38.0;
10, 128, 28, 37.9];
>> coreTemperature = VitalSigns(:,end);
fahrenheitTemp = (coreTemperature*9/5) + 32;
```

Before we incorporate the new temperature values into the matrix, let us create a copy of the 'VitalSigns' matrix:

```
>> modifiedData = VitalSigns;
```

Now, let us replace the appropriate temperature column with the column containing the new values:

```
>> modifiedData(:,end) = fahrenheitTemp
modifiedData =
         0   64.0000   10.0000   98.2400
    2.0000   78.0000   15.0000   98.6000
    5.0000  120.0000   20.0000   99.5000
    8.0000  130.0000   30.0000  100.4000
   10.0000  128.0000   28.0000  100.2200
```

At this stage, we have our new data matrix, and we can write it to a text file, either with or without a header.

But what if instead of overwriting the temperature data, we want to 'snap on' an extra column onto the existing data matrix? This is achieved quite easily by concatenating the new column at the end of the original matrix:

```
>> expandedData = [VitalSigns, fahrenheitTemp]

expandedData =

         0   64.0000   10.0000   36.8000   98.2400
    2.0000   78.0000   15.0000   37.0000   98.6000
    5.0000  120.0000   20.0000   37.5000   99.5000
    8.0000  130.0000   30.0000   38.0000  100.4000
   10.0000  128.0000   28.0000   37.9000  100.2200
```

Exporting without headers

When exporting without headers, we simply need to specify three inputs: the name of the file we want to create, the matrix we want to write to the file, and the symbol used for separating the numbers.

It is quite common to read/write text files containing numbers that are separated by either a comma (‘,’), or a tab (‘\t’). The syntax for writing out a file with these ‘demiliters’ is shown below:

```
>> dlmwrite('commaSeparated.txt', modifiedData, ',')

>> dlmwrite('tabSeparated.txt', expandedData, '\t')
```

You can use any symbol you want as a delimiter. However, I would advise you to use either commas, tabs, or spaces.

With headers

Exporting data with headers included (i.e., column headings to indicate the names of the variables) is slightly more involved. However, once you have the data matrix and the column names you want to write to file, you can use the following commands as a template.

```
>> columnNames = {'Time'; 'Pulse'; 'BreathRate'; 'TempCentigrade'; 'TempFahrenheit'}
T = array2table(expandedData,'VariableNames',columnNames)
writetable(T, 'dataWithHeaders.txt', 'Delimiter', ',')
columnNames =
    'Time'
    'Pulse'
    'BreathRate'
    'TempCentigrade'
    'TempFahrenheit'
T =
```

Time	Pulse	BreathRate	TempCentigrade	TempFahrenheit
0	64	10	36.8	98.24
2	78	15	37	98.6
5	120	20	37.5	99.5
8	130	30	38	100.4
10	128	28	37.9	100.22

Now, you should have a text file called ‘dataWithHeaders.txt’ in your working directory, which you can examine using any text editor of your choice.

You can even import back the data we just wrote, using the commands mentioned in the Section on Importing Data:

```
>> myData = importdata('dataWithHeaders.txt')

expandedVitalSigns = myData.data

myData =

            data: [5x5 double]
        textdata: {'Time'  'Pulse'  'BreathRate'  'TempCentigrade'  'TempFahrenheit'}
    colheaders: {'Time'  'Pulse'  'BreathRate'  'TempCentigrade'  'TempFahrenheit'}

expandedVitalSigns =

         0   64.0000   10.0000   36.8000   98.2400
    2.0000   78.0000   15.0000   37.0000   98.6000
    5.0000  120.0000   20.0000   37.5000   99.5000
    8.0000  130.0000   30.0000   38.0000  100.4000
   10.0000  128.0000   28.0000   37.9000  100.2200
```

Understanding the difference between 'a=b' and 'a==b'

This is a minor issue (relevant for almost all programming languages) which confused me for a while when I was a beginner. The professors explained that 'one is an assignment statement, and the other is a comparison statement', but it did not really click. We will quickly look at how these two statements differ in case you are puzzled.

When we want to create a new variable (say, the number of teams playing in a particular soccer league in a particular year), and set its value, we use the '=' operator:

```
>> teamsIn2015 = 10;

>> teamsIn2014 = 8;
```

Now suppose you want to check whether the same number of teams played in 2014 and 2015. For this, we would use the '==' operator

```
>> teamsIn2014 == teamsIn2015

ans =

     0
```

Matlab says 'false'! The number of teams playing in 2014 is not 'equalequal' to the number of teams playing in 2015.The professors were correct after all in explaining that 'one is an assignment statement, and the other is a comparison statement'!

We could even have ‘compared’ whether one number is ‘bigger/smaller’, or ‘bigger/smaller and equal to’ the other number. Such comparison statements are extremely useful in constructing if-statements.

```
>> teamsIn2014 <= teamsIn2015

teamsIn2014 < teamsIn2015

teamsIn2014 > teamsIn2015

ans =

     1

ans =

     1

ans =

     0
```

THREADING IT ALL TOGETHER

The Problem

Let us consider a simple problem that will let us put to use most of the commands and concepts that we have learned. We will compare the physical performance of 3 different athletes and rank them according to their fitness.

Suppose that we are given three text files (‘athlete1.txt’, ‘athlete2.txt’, and ‘athlete3.txt’) containing data about the vital signs of three different athletes who are exercising. Our task is to figure out which athlete is the fittest, by assigning scores to each of them, based on the values we read in from the text files.

The instructions are as follows: “Find the fittest athlete out of the three for whom you are given data. At each time instant, compare the core temperature, breathing rate, and pulse of each of the three athletes. Assume that lower values of these metrics correspond to a higher level of fitness.”

You must understand that there is no single way of accomplishing any particular task. However, we will consider the simplest way of coding up a solution for this problem.

Formulating Our Solution

Before we jump to writing code, we must decide on a few things about our solution strategy. Our problem-statement leaves most of the details of how we come up with a solution to us! This will generally be the case anytime we try to solve a practical problem.

First of all, we need to decide how we will rank our athletes. One possibility is to assign 1 point to the athlete with the fastest pulse (lowest point awarded, since a faster heartbeat-rate signifies lower fitness), 2 points to the athlete with the second-fastest pulse, and 3 points to the athlete with the slowest pulse (fittest individual). We can use this same strategy for core-temperature and breathing rate, since lower values for these metrics indicate higher fitness. We will keep track of the athletes’ total by adding up points awarded at each step, and at the very end, declare the athlete with the most points the winner!

Loading the Data

Let us start with a new Matlab script, and call it ‘findFittest.m’. Assume that the three text files given to us look as follows:

'athlete1.txt' contains:

Time,Pulse,BreathRate,Temp

0,64,10,36.8

2,78,15,37

5,120,20,37.5

8,130,30,38 10,128,28,37.9

'athlete2.txt' contains:

Time,Pulse,BreathRate,Temp

0,68,12,36.8

2,82,17,37.3

5,123,24,37.6

8,135,35,38.1

10,140,36,38.0

'athlete3.txt' contains:

0,60,8,36.8

2,75,20,36.9

5,125,22,37.6

8,128,28,37.8

10,126,30,37.9

Let us load these data files into Matlab:

```
clear
close all
clc
```

```
ath1 = importdata('athlete1.txt');
athlete1Table = ath1.data;

ath2 = importdata('athlete2.txt');
athlete2Table = ath2.data;

athlete3Table = importdata('athlete3.txt');
```

Notice that we did not have to use the extra instruction 'ath3.data' for the third athlete, since the third file does not contain any headers (click on Importing Data to go to the section with the relevant details). Now, there are three 2D matrices ready for us to manipulate.

Translating from English to algorithm

Now, we must decide how exactly we will implement the code that will rank the athletes in order of fitness. Think about how you would approach the problem if you were to solve it using just pen and paper.

You might decide to look at each row in the data-table (Athlete 1's data-table shown below), look at the pulse first, compare the pulse at this instance for all three athletes, and award them points.

Time [minutes]	Pulse [beats/min]	Respiratory Rate [breaths/min]	Temperature
0	64	10	36.8
2	78	15	37.0
5	120	20	37.5
8	130	30	38.0
10	128	28	37.9

Next, you might compare the respiratory rate for the three athletes at this time instance (i.e., a particular row in the tables), and award more points.

Then you would do the same for the core temperature.

After this, you would move on to the next row, and repeat the process of assigning points.

Learning how to translate this sort of solution-strategy into working code is often the most difficult challenge that new programmers face. We will learn how to do this in the next few pages.

We decided that we would look through each row in the three tables when comparing the three athletes. For this, we must write a for-loop (see the section 'For Loops and if conditionals' for more details on this topic) that will visit each of the rows in the tables. Please note again that we will assume that the three tables we are given are of the same shape and size, which is true for the data in the three .txt files given to us for this specific problem.

```
for indRow = 1:size(athlete1Table, 1)

    …

end
```

This is the skeleton of what our code is to become! We will fill it in, one-step at a time. However, even experienced programmers must start with this first step.

Now, think about what we should do next. Right now, our code skeleton will let the computer loop through each row in a table. Therefore, we can store the rows for the three different athletes in three different variables as follows (code in italics/grey is repeated exactly from the previous step):

```
for indRow = 1:size(athlete1Table, 1)

    % Extract each row at a time from the 3 data-tables
    athlete1Row = athlete1Table(indRow, :);
    athlete2Row = athlete2Table(indRow, :);
    athlete3Row = athlete3Table(indRow, :);

  …

end
```

As the next step for our pen and paper solution, we would look at each of the three variables stored in each row, and compare them. Let us write the code to do exactly this in Matlab.

We must write another for-loop to individually look at each of the three variables (pulse, breathing rate, and temperature) stored in each row. Keep in mind that at this moment, we are writing the simplest code that can accomplish our task.

```
for indrow = 1:size(athlete1Table, 1)

    % Extract each row at a time from the 3 data-tables
    athlete1Row = athlete1Table(indRow, :);
    athlete2Row = athlete2Table(indRow, :);
    athlete3Row = athlete3Table(indRow, :);

    % Write a 'nested for-loop' to extract each element stored in every row
    for indCol = 1:length(athlete1Row)
        athlete1Data = athlete1Row(indCol);
        athlete2Data = athlete2Row(indCol);
        athlete3Data = athlete3Row(indCol);
        …
    end

end
```

We are almost there! Now, we must write the comparison statements, which will decide the amount of points awarded to each athlete. In addition to doing this, we must create a variable to keep track of the total points of each of the athletes, and initialize these to zero.

> Programming pitfall: Note that the initialization-to-zero must be kept outside of all the for-loops, otherwise the athletes' points will be set to zero at every iteration. Forgetting to do this is a common newbie mistake.

Before we proceed any further, you might have realized that the first column in each row contains the time values, which we must exclude for our data-comparison purposes. I did not catch this little detail until I actually ran the program, looking carefully through the output of each line! We can include commands to delete the first column of each of the three matrices we have loaded:

```
athlete1Table(:,1) = [];
athlete2Table(:,1) = [];
athlete3Table(:,1) = [];
athletesTotal = [0, 0, 0];
for indRow = 1:size(athlete1Table, 1)

        % Extract each row at a time from the 3 data-tables
        athlete1Row = athlete1Table(indRow, :);
        athlete2Row = athlete2Table(indRow, :);
        athlete3Row = athlete3Table(indRow, :);

        % Write a 'nested for-loop' to extract each element stored in every row
        for indCol = 1:length(athlete1Row)

      athlete1Data = athlete1Row(indCol);
      athlete2Data = athlete2Row(indCol);
      athlete3Data = athlete3Row(indCol);
      % Find which athlete has the lowest value for the variable being inspected, and
      who has the highest value
      [minValue, minLocation] = min([athlete1Data, athlete2Data, athlete3Data]);
      [maxValue, maxLocation] = max([athlete1Data, athlete2Data, athlete3Data]);
      % Give the athlete with the lowest value 3 points, and the athlete with the
      highest value 1 point
      athletesTotal(minLocation) = athletesTotal(minLocation) + 3;
      athletesTotal(maxLocation) = athletesTotal(maxLocation) + 1;
          % Identify the leftover athlete, and give him 2 points
       …
   end
end
```

Now comes the slightly tricky part! How do we identify the leftover athlete, i.e., the one with the intermediate value for the variable being examined?

It is easy for us human beings to identify which number has not been picked from a list, but doing this is not straightforward for a computer.

The crudest way to accomplish this is as follows: create a temporary vector [1, 2, 3], and remove the elements at 'minLocation' and 'maxLocation' from this array.
The remaining element will tell us the number of the leftover athlete.

```
athlete1Table(:,1) = [];
athlete2Table(:,1) = [];
athlete3Table(:,1) = [];
athletesTotal = [0, 0, 0];
for indRow = 1:size(athlete1Table, 1)

    % Extract each row at a time from the 3 data-tables
    athlete1Row = athlete1Table(indRow, :);
    athlete2Row = athlete2Table(indRow, :);
    athlete3Row = athlete3Table(indRow, :);

    % Write a 'nested for-loop' to extract each element stored in every row
    for indCol = 1:length(athlete1Row)

        athlete1Data = athlete1Row(indCol);
        athlete2Data = athlete2Row(indCol);
        athlete3Data = athlete3Row(indCol);
```

```
        % Find which athlete has the lowest value for the variable being inspected, and who
        has the highest value
        [minValue, minLocation] = min([athlete1Data, athlete2Data, athlete3Data]);
        [maxValue, maxLocation] = max([athlete1Data, athlete2Data, athlete3Data]);

        % Give the athlete with the lowest value 3 points, and the athlete with the
        highest value 1 point
        athletesTotal(minLocation) = athletesTotal(minLocation) + 3;
        athletesTotal(maxLocation) = athletesTotal(maxLocation) + 1;
        % Identify the leftover athlete, and give him 2 points
        tempVec = 1:3;
        tempVec([minLocation, maxLocation]) = []
        athletesTotal(tempVec) = athletesTotal(tempVec) + 2;

    end
end
```

After having written this somewhat clunky algorithm, I realized that there is a much cleaner way to do this! We could have sorted the variable values, and this would tell us exactly which athlete deserves 1,2, or 3 points:

```
athlete1Table(:,1) = [];
athlete2Table(:,1) = [];
athlete3Table(:,1) = [];

athletesTotal = [0, 0, 0];
for indRow = 1:size(athlete1Table, 1)

        % Extract each row at a time from the 3 data-tables
        athlete1Row = athlete1Table(indRow, :);
        athlete2Row = athlete2Table(indRow, :);
        athlete3Row = athlete3Table(indRow, :);
```

```
    % Write a 'nested for-loop' to extract each element stored in every row
    for indCol = 1:length(athlete1Row)

    athlete1Data = athlete1Row(indCol);
    athlete2Data = athlete2Row(indCol);
    athlete3Data = athlete3Row(indCol);

% Find which athlete has the lowest value for the variable being inspected, and who has the highest value
[sortedValue, sortingIndex] = sort([athlete1Data, athlete2Data, athlete3Data]);

% The values in 'sortingIndex' now contain the index locations with the smallest, the intermediate, and the largest values, in order

% Give the athlete with the lowest value 3 points, the athlete with the intermediate value 2 points, and the athlete with the highest value 1 point

        athletesTotal(sortingIndex(1)) = athletesTotal(sortingIndex(1)) + 3;
        athletesTotal(sortingIndex(2)) = athletesTotal(sortingIndex(2)) + 2;
        athletesTotal(sortingIndex(3)) = athletesTotal(sortingIndex(3)) + 1;


    end

end
```

And we are done! The last part of the task is to figure out who won. We can do this by using the 'sort' command once more! The entire script to perform our task is shown below.

You must store and run it in the same directory where you have stored the 'athlete1.txt', 'athlete2.txt', and 'athlete3.txt' files.

```
clear
close all
clc

ath1 = importdata('athlete1.txt');
athlete1Table = ath1.data;

ath2 = importdata('athlete2.txt');
athlete2Table = ath2.data;
athlete3Table = importdata('athlete3.txt');
athlete1Table(:,1) = [];
athlete2Table(:,1) = [];
athlete3Table(:,1) = [];
athletesTotal = [0, 0, 0];
for indRow = 1:size(athlete1Table, 1)

    % Extract each row at a time from the 3 data-tables
    athlete1Row = athlete1Table(indRow, :);
    athlete2Row = athlete2Table(indRow, :);
    athlete3Row = athlete3Table(indRow, :);

    % Write a ‘nested for-loop’ to extract each element stored in every row
    for indCol = 1:length(athlete1Row)

        athlete1Data = athlete1Row(indCol);
        athlete2Data = athlete2Row(indCol);
        athlete3Data = athlete3Row(indCol);
```

```
        % Find which athlete has the lowest value for the variable being inspected, and
        who has the highest value
        [sortedValue, sortingIndex] = sort([athlete1Data, athlete2Data, athlete3Data]);

        % The values in 'sortingIndex' now contain the index locations with the
        smallest, the intermediate, and the largest values, in order

        % Give the athlete with the lowest value 3 points, the athlete with the
        intermediate value 2 points, and the athlete with the highest value 1 point


        athletesTotal(sortingIndex(1)) = athletesTotal(sortingIndex(1)) + 3;

        athletesTotal(sortingIndex(2)) = athletesTotal(sortingIndex(2)) + 2;

        athletesTotal(sortingIndex(3)) = athletesTotal(sortingIndex(3)) + 1;

    end


end

[sortedPoints, sortedAthletes] = sort(athletesTotal)
```

The result you will get from running this script should look as follows:

```
sortedPoints =

   19    34    37

sortedAthletes =

    2     3     1
```

Therefore, athlete1 has the highest number of points (at 37), with athlete3 close behind (at 34 points), and athlete2 really seems to be an average-Joe, lagging far behind the other two (with 19 points)!

You might have noticed by now that we did not write our code from 'top to bottom', but rather, filled in the gaps as we walked through our logic of solving the problem. This is usually how you should write code. You might also have realized that even experienced programmers get things wrongs and write inefficient code quite frequently! The trick to recover from your stumbles, is to always write short chunks of code, and to test these chunks each step of the way as you build up your program.

When testing your scripts, you should start at the very top, and remove the ';' at the end of a line, with all the other line outputs still suppressed (i.e., the other lines have a ';' at the end). Then run the program, and make sure that the line you are looking at does

exactly what you want it to do. Then suppress the output for this line (reintroduce the ';' at the end), and unsuppress the next line. Keep doing this until you make sure that each of the lines in the script does exactly what you would expect if you were to solve the problem using pen and paper. This may seem unnecessary to you now, but catching bugs early on is key to saving a lot of time later on!

Note that we did not consider what should happen when more than one athlete has the same value for the variable being examined. For instance, both athlete1 and athlete3 have the same core temperature (36.8 degrees C) at time = 0. As an exercise, you should modify our script to account for this (for instance, by including extra if-statements or for-loops), and assign equal points to all the athletes with the same values at a particular instance.

Summary

Having written your first program, you probably realize that programming is mostly logic-based: if you can figure out a way to solve a problem using pen and paper, all that remains is translating your thoughts into Matlab syntax. You can do far more interesting things with the various dataset we have used here, but the purpose of this book is to introduce you to the basic syntax you will use most often in Matlab, and to get you comfortable with the concept of programming in general!

PARTING WORDS

We have not even begun to scratch the surface of all that is possible with Matlab, but what you have learned will serve you well for laying a strong foundation. There is much more you can do with data analysis, 2D/3D graphics, digital signal processing, image segmentation and processing, and even building GUIs (Graphical User Interface) within Matlab!

There is a plethora of free information available on the internet for this sort of stuff, and my hope is that the basics outlined in this book will let you venture out on your own to more advanced topics. Hopefully, after having gone through the book, you will be able to filter out the useful stuff from the abundance of information available to you.

If you have enjoyed the material in this short introductory book, and are raring to go for more, be sure to check out other 'Matlab for newbies' books, which I hope to finish writing at some point in the future!

If you find mistakes/typos in this book, or have suggestions for improving it, please do not hesitate to contact me at vermas@fau.edu.

In addition, do not forget to create, explore, and have fun!

About the Author

The author wanted to become an astronaut for as long as he can remember. Alas, gravity proved to be too strong a force! He settled for the next best thing, and started studying Aerospace Engineering at the Georgia Institute of Technology (Georgia Tech) in 2006. After dabbling in the dark arts (experimental research) for a couple of years, the author decided to figure out what computational physics was all about! The fact that he had used Matlab to control supersonic wind-tunnel experiments probably played some part in this decision. Soon, he became hooked on the idea of solving strange, uncooperative mathematical equations using some of the most powerful supercomputers in the world.

A hasty (and surprisingly painful) Master's degree from the California Institute of Technology (Caltech) in 2010, followed by a delightfully wonderful four-year-long Doctorate degree, gave him ample opportunity to make computers sing to his tune! It is the author's sincere hope that after reading this short book, you too are infected by the computing bug, and go on to do wonderful things for the planet!

The author is currently an Assistant Professor in the Department of Ocean & Mechanical Engineering, at the Florida Atlantic University (FAU).

All plots in this book were created using GNU Octave.