Chapter 1

Computational Science

1.1 What is Computational Science?

Computational Science is the development and use of software tools as part of a scientific inquiry. It is also sometimes called Scientific Computing.

Computational Science is an increasingly important part of Physics, Chemistry and other natural sciences, Economics and Sociology and other social sciencs.

This class covers three kinds of topics

Programming: How to use Matlab and write Matlab programs.

Numerical methods: Techniques for getting approximate answers to problems without exact analytic solutions.

Modeling: Abstract models of real-world systems.

Abstract models are central to our understanding of the physical world. To say that a model is "abstract" is to say that some real-world details have been omitted. By simplifying the model, we make it possible to derive the behavior of the system analytically, or simulate it using a computer.

For example, in mechanics it is often useful to model real mechanisms using abstractions like frictionless pulleys and perfect springs. Of course, neither of these things exists. The assumption we are making is that the friction of a real pulley, or the non-linearity of a real spring, has a small effect on the behavior of the system (compared to the size of the behavior we are interested in).

If that assumption turns out to be right, then any error in the result will be tolerable. If not, then we might have to refine the model, adding back some of the details we removed.

In general, Applied Mathematics concerns itself with models that can be solved analytically and Computational Science deals with models that have to be solved numerically. The nice thing about analytical solutions is that they are exact and general. The nice thing about numerical methods is that you can apply them to a wider range of problems. Numerical methods make it possible to choose models that are more realistic, because they include more real-world details.

1.2 Why program?

Most simulations come in the form of computer programs. In this book we will look at programs written in MATLAB, which is both a computer language and an interactive tool designed for numerical simulations.

A program is a sequence of instructions that specifies how to perform a computation. The instructions (or commands, or statements) look different in different programming languages, but there are a few basic functions that appear in just about every language:

input: Get data from the keyboard, or a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

testing: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

That's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look more or less like these. One way to define programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these simple instructions.

1.3 The MATLAB interpreter

Most MATLAB programming is done interactively using the MATLAB interpreter. The interpreter performs the following steps over and over:

input: Prompt the user to input a statement.

execute: Interpret the statement and do what it says.

display: Show the user the result of the computation.

In the MATLAB command window, the characters `>>` are the **prompt**. They indicate that you can type and execute a **statement**.

The most basic kind of statement MATLAB can execute is a mathematical **expression**. For example, if you type `7 + 13`,

```
>> 7 + 13
ans = 20
```

MATLAB prints the result. I have removed some of the space from the output format.

The other mathematical operations work the same way, using `-` for subtraction, `*` for multiplication, and `/` for division.

MATLAB can also evaluate pretty much any mathematical function you have heard of, like sin, cos, log, absolute value and square root. In many cases MATLAB expressions look just like math.

```
>> sin (1)
ans = 0.8415

>> log (10)
ans = 2.3026
```

One difference is that the parenthese around the argument are required. In mathematics you can write $sinx$, but in MATLAB you have to write `sin (x)`.

In some cases it is inconvenient to express mathematical notation using the keys on the keyboad, so MATLAB uses alternative notation. For example, we have to use the `sqrt` function because it is hard to draw the square root symbol with the mouse.

To raise a number to a power, as in y^x you have to write `y^x`, but for e^x you would usually write `exp (x)`. The function `log` uses base e; there is another function called `log10` that uses base 10.

One of the first exercises for the class is to take mathematical expressions and translate them into MATLAB.

1.4 What is debugging?

Programming is a complex process, and since it is done by humans, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.4.1 Syntax errors

The interpreter can only evaluate a statement or program if it is syntactically correct. **Syntax** refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

MATLAB is not so forgiving. If there is a single syntax error anywhere in a statement, the interpreter prints an error message and refuses to execute the statement.

A common error people make when they use MATLAB is to use the implicit notation for multiplication. In mathematical notation, $3(2-1)$ implies that the expression $2-1$ should be multiplied by 3. In MATLAB that gets you an error message:

```
> 3 (2 - 1)
??? 3 (2 - 1)
      |
Error: Missing operator, comma, or semicolon.
```

The error message indicates what MATLAB thinks in wrong, and the vertical line indicates where MATLAB thinks the error is. Although this information is sometimes useful, it can be misleading.

During the first few weeks of your programming career, you will probably spend a lot of time dealing with syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.4.2 Run-time errors

The second type of error is a **run-time error**, so-called because the error does not appear until a statement is evaluated.

One kind of run-time error is a **warning** indicating that you have done something questionable like like division by zero.

```
>> 1/0
Warning: Divide by zero.
ans = Inf
```

The result in this case, infinity, is correct, but the warning suggests that there might be something wrong, since infinity is often a problematic value in a computation.

Other run-time errors force MATLAB to quit what it is doing without producing an answer. At this point, though, we don't really know enough to get into trouble. We'll see more run-time errors later.

1.4.3 Logic errors and semantics

The third type of error is the **logical** or **semantic** error. If there is a logical error in your program, it will run successfully, in the sense that MATLAB will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

1.4.4 Approximation errors

Even if your program runs successfully and does what you want it to do, the result might not be exactly right. It might be an approximation of the right answer. As a simple example, if you divide 2 by 3, you get

```
>> 2/3
ans = 0.6667
```

The actual answer has a lot more sixes! The problem here is that the computer cannot represent the actual answer exactly, so it has to round off.

In more complicated computations, approximation errors occur because the input values are approximate, because the computation introduces errors, or because the computation stops when the answer is close enough.

The definition of "close enough" depends on the nature of the problem and what the answer will be used for. For each of the numerical methods we work with, we will discuss what kinds of errors can occur and how to make sure the answer is close enough.

We will see some cases where approximation errors are intrinisic to the problem, so there is nothing we can do, and other cases where we can change the program to improve the accuracy of the result.

1.5 Experimental debugging

One of the most important skills you will acquire in this class is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (from A. Conan Doyle's *The Sign of Four*).

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does *something*, and make small modifications, debugging them as you go, so that you always have a working program.

1.6 Formal and natural languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, mathematical notation is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

As I mentioned before, formal languages tend to have strict rules about syntax. For example, $3+3 = 6$ is a syntactically correct mathematical statement, but $3 = +6\$$ is not. Also, H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax and semantics—there are many differences.

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.7 Scripts

Using an interpreter to write programs is often convenient, and it makes it easy to debug, but once you have written a program you usually want to make a permanent copy. In MATLAB there are two ways to do that, script files and function files.

A script file is a list of statements stored in a file with a name that ends in `.m` (for MATLAB). You can execute the commands by typing the name of the script (without the `.m`).

For example, if there is a file with the name `fred.m` that contains these statements

```
4/5 + 1/3
4/5 * 1/3
```

then we can execute the script by typing

```
>> fred
ans = 1.1333
ans = 0.2667
```

and MATLAB displays the results.

1.8 Scientific notation

If you have taken astronomy, you have seen scientific notation, which is a convenient way to write large (and very small) numbers. The format is like $2.14 \cdot 10^8$, which means $2.14 \cdot 100,000,000$, which is $214,000,000$. The number 2.14 is called the mantissa and 8 is the exponent.

One nice thing about scientific notation is that it is concise, since we don't have to write lots of zeros. Another nice thing is that it makes it clear how many of the digits in a number are significant. If I write that the speed of light in a fiber optic cable is $2.14 \cdot 10^8$ meters per second, the reader understands that the true value might be between 2.135 and 2.145.

If the result of a computation is particularly large or small, MATLAB reports it in scientific notation.

```
>> tan (3.141)
ans = -5.9265e-04
```

In this format, the letter `e` stands for exponent, and it separates the mantissa from the exponent. So `-5.9265e-04` represent $-5.9265 \cdot 10^-4$, which is .00059265.

You can also use this format to input numbers, so you could write

```
>> c = 2.14e8
c = 214000000
```

In this case MATLAB decided to print the result in normal notation. There are a number of options you can use to control MATLAB's output format. Type `help format` for more information.

1.9 Variables

The previous example is an **assignment statement**, because it assigns a value (`2.14e8`) to a variable. A variable is a named location that can store a value. Once you assign a value to a variable you can use it in an expression anywhere you would use a number.

```
>> c = 2.14e8
>> distance = 4000
>> time = distance / c
time = 1.8692e-05
```

Assigning values to variables is useful for a number of reasons, but for now the most important reason it that it makes the program easier to read. You probably figured out that the previous calculation computed the time it takes a signal to traverse 4km of fiber optic cable.

By convention, variable names in MATLAB are all lower case. You can also use numbers as part of a variable name, but the first character has to be a letter.

If you use a name with more than one word, you can separate words with the underscore character, as in `a_long_variable_name`. But don't go too crazy; names can only be 31 characters long.

Some names are **keywords**, meaning that they are part of the MATLAB langauge and cannot be used as variable names. These names are

```
for  end  if  while  function  return  elseif  case  otherwise
switch  continue  else  try  catch  global  persistent  break
```

Fortunately, most of these are words you probably wouldn't have chosen anyway.

MATLAB provides a number of built-in variables that contain special values like π and infinity:

```
>> pi
ans = 3.1416
>> inf
ans = Inf
```

The other special variables are

```
ans  beep  eps  nan  i  j  nargin  nargout
realmin  realmax  bitmax  varargin  varargout
```

In general you should not use these names for your variables, either. The exceptions are `i` and `j` which are initially defined to be the square root of -1, but are often used for other purposes.

1.10 Help

This book demonstrates the most basic use of MATLAB's commands. It is not intended to be a reference manual for MATLAB. Fortunately, there *is* a reference manual for MATLAB, and it is even available online. Whenever you start working with a new command, you should get in the habit of reading its documentation.

The `help` command is one way to get this documentation. For example, `help format` prints information about the `format` command.

```
>> help format

 FORMAT Set output format.
    All computations in MATLAB are done in double precision.
    FORMAT may be used to switch between different output
    display formats as follows...
```

Also, the command `help` all by itself lists the topics on which help is available. `helpwin` is the same as `help` except that the reply appears in a new window, which is sometimes more convenient.

Finally, the command `lookfor` searches the documentation for commands that contain a given word in their description. For example, `lookfor precision` prints descriptions of 14 commands that mention the word precision. Unfortunately, `lookfor` often takes a long time to run.