Computational Science

Allen B. Downey

Computational Science

This book was typeset by the author using LaTeX and dvips, which are both free, open-source programs.

Contents

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 vetical 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. But we will.

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 many of the numerical methods we will look at, 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 its accuracy.

1.4.5 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.

1.5 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.

1.6 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.

1.7 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.

1.8 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.

1.9 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.

Chapter 2

Fibonacci

2.1 Errors

Since we are going to be dealing with approximations, we should think for a second about errors. There are at least two ways of thinking about errors, called **absolute** and `relative`.

An absolute error is just the difference between the correct value and the approximation. We usually write the magnitude of the error, ignoring its sign, because it doesn't matter whether the appoximation is too high or too low.

For example, we might want to estimate 9! using the formula $\sqrt{18\pi}(9/e)^9$. The exact answer is $9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 362,880$. The approximation is $359,536.87$. The absolute error is 3,343.13.

At first glance, that sounds like a lot—we're off by three thousand—but it is worth taking into account the size of the thing we are estimating. For example, $3000 matters a lot if we are talking about my annual salary, but not at all if we are talking about the national debt.

A natural way to handle this problem is to use `relative error`, which is the error expressed as a fraction (or percentage) of the exact value. In this case, we would divide the error by 362,880, yielding .00921, which is just less than 1%. For many purposes, being off by 1% is good enough.

2.2 Numbers

In mathematics we seldom give much thought to the amount of space a number takes up. As numbers get bigger and bigger, it takes more and more digits to write them down, but in general that is not a problem. In fact, people often use the number of digits in a number as a rough estimate of its size, as in "a six-digit salary."

In computer arithmetic, we are limited by the ability of the hardware to deal with large numbers. There are usually a fixed number of digits available to record each number, and that limits the size of the numbers we can compute.

As a small example, imagine that we have enough space to record 4 digits for every number we want to represent. One way to use those four digits it to interpret them as an integer between 0000 and 9999. There are 10,000 such integers, evenly spaced on the number line.

Of course such a system has limitations. First, the range is limited; we cannot represent numbers bigger than 9999. Also, even in the range, there are numbers we can't represent, like fractions.

The first limitation leads to a condition called **overflow**, which means that the result of an operation is not in the range. For example, in 4-digit arithmetic, we cannot compute `9999 + 1` or `100 * 100`. In MATLAB an operation that overflows produces the special value `Inf`. For example the `factorial` function often produces big values:

```
>> factorial (100)
ans = 9.3326e+157
>> factorial (200)
ans = Inf
```

The value of 100! can be represented, but the value of 200! cannot.

The second limitation leads to an error called `roundoff`, which means that the result of an operation cannot be represented exactly, so we have to round off to the nearest representable number. We have already see one example, where the result of dividing 2 by 3 is rounded off to 0.6667. Whenever you perform an operation, there is a chance that the result cann be represented exactly. In general you should assume that roundoff occurs after every operation.

In most cases the errors that are introduced by roundoff are small and have no practical impact. In other cases, roundoff errors can accumulate and yield results that are wildly inaccurate.

Unfortunately, the computer seldom knows whether a result is accurate. It if usually happy to display incorrect results with no warning. It is up to the user to know whether a given result can be trusted, which requires some knowledge about the nature of the problem being solved and the numerical method being used. This class is intended to convey that kind of knowledge.

In integer arithmetic, the biggest error you can get from a single operation is 0.5. That means that the absolute error is pretty much the same throughout the range of representable numbers. The relative error is not. If the exact answer is 1 and the abolute error is 0.5, then the relative error is 50%. But if the exact answer is 9999, then the maximum relative error is about 0.005

2.3 Floating-point numbers

Most computers provide an alternate way to represent numbers that is called `floating-point`. Floating-point notation is very similar to scientific notation, except that we have a fixed number of digits in the mantissa and in the exponent.

Looking again at the 4-digit format, we could use one digit as an exponent and the other three digits as a mantissa. Thus, we would understand that the number `123 5` means $1.23 \cdot 10^5$, or 123,000. Using this format, the largest number we can represent is 9,990,000,000, which is much larger than the largest integer we can represent with the same number of digits.

The price we pay for the increased range is that the space between the representable numbers has stretched. The second largest number we can represent is 9,980,000,000, which leaves a gap of 10,000,000. So the largest possible roundoff error is 5,000,000!

Again, that seems like a lot, until we consider the relative error. Being off by 5 million is not so bad if the exact value is 9,985 million. The relative error is only 0.05%.

Interestingly, the maximum relative error is the same over almost the entire range. For example, if the number we want to represent is 1000.5, and the best we can do is 1000, then the relative error is 0.05% again.

At the low end, the relative errors can be significant. If we want to represent the number 1.5 and the best we can do is 1, the relative error is 33%.

In the worst case, if the number we want to represent is less than 0.5, then we round down to zero and the relative error is 100%. Rounding down to zero is also called `underflow`. If a computation causes an underflow, the result is meaningless, so most systems will report a run-time error.

2.4 MATLAB numbers

MATLAB performs all computations using floating-point numbers, even when the operands are integers. Fortunately it uses more than 4 digits, so most of the time the errors are small or zero. If you use the command `format long`, MATLAB displays all the digits of the mantissa:

```
>> format long
>> 2/3
ans = 0.66666666666667
```

The number of significant digits is 15 or 16. This number represents the **precision** of the result (how precisely it is expressed). That is not the same thing as the **accuracy**, which determines how many of those digits are correct.

The range of representable numbers is roughly from 10^{-300} to 10^{300}. As an exercise, try to figure out the range more accurately.

In the previous section I said that most systems report underflow as an error. MATLAB does not, which can be problematic. For example, if you compute

```
>> x = 1e300
x = 1.0000e+300
>> x / x^2
ans = 0
```

you get the wrong answer and no warning. I think it would be better for MATLAB to say, "I can't perform this computation" than to do it wrong.

In this case, all the operands are representable, and the result is representable, but the computation underflows. We can rewrite the same expression to get the correct result:

```
>> x / x / x
ans = 1.0000e-300
```

This example demonstrates one of the other properties of floating-point arithmetic: it is neither associative nor commutative. For example, in real arithmetic, it doesn't matter whether you compute `1 + (2 + 3)` or `(1 + 2) + 3` or `(3 + 2) + 1`. In floating-point arithmetic, it does (at least sometimes).

Faires and Burden give several examples of computations that are equivalent in real arithmetic but that yield different results in floating-point arithmetic (pages 20–23 and exercise 6 on page 26).

2.5 Nested functions

If you have seen a mathematical expression like $\sqrt(\sin(x))$, you have seen a nested function. To evaluate something like this you (or the MATLAB interpreter) start from the innermost parentheses and work your way out. In this case you would compute the square root of x and then apply the sin function to the result.

In MATLAB, as in mathematics, you can nest functions and other expressions with arbirary depth. So, for example, there is nothing wrong with the expression:

```
((x - theta) * sqrt (2 *pi) * sigma) ^ exp (-1/2 * (log (x - theta) - zeta)^2 / sigma^2)
```

except that it is big and hairy.

If you write something like this, you might want to break it into multiple lines. Just type `...` at the end of the first line and continue on the next.

2.6 Temporary variables

An even better way to write big, hairy expressions is to break the process into several steps and use temporary variables to store intermediate results.

For example, I would rewrite the previous example like this:

```
shiftx = x - theta
base = shiftx * sqrt (2 *pi) * sigma
temp = (log (shiftx) - zeta) / sigma
exponent = -1/2 * temp^2
result = base ^ exp (exponent)
```

I chose names for the variables that explain their role in the computation. shiftx is the value of x shifted by the value of theta. At the end of the computation, it should be no surprise that base gets raised to a power and exponent gets exponentiated. I couldn't think of a good name for temp, so I called it temp.

Choosing informative names makes the code easier to understand. It can also make the code simpler, and sometimes faster. For example, the subexpression x - theta appears twice in the original expression. By computing it once and assigning it to a variable, we make the code smaller, and we might save MATLAB from performing the subtraction twice, assuming that MATLAB wouldn't have figured that out for itself.

2.7 Function files

In the previous chapter we saw a way to save MATLAB programs in script files and invoke them later. Function files are similar to script files in the sense that they store commands for later execution.

The also provide a very useful capability: you can add new functions to MATLAB. For example, we can define a function named square so that square(x) computes the value of x^2.

We have to create a file named square.m that contains the following:

```
function y = square (x)
% SQUARE  Computes x raised to the second power.
y = x^2
```

The first word of a function file has to be function, which distinguishes it from a script file.

The first line contains the function declaration, which includes the name of the function, square, the name of the input variable x, and the name of the output variable, y.

The second line is a comment that describes, in English, what this function does. A comment can appear anywhere in the program. It always starts with a % character and goes to the end of the line. You can put anything you want in a comment; it doesn't affect the behavior of the program.

The last line is an assignment statement that computes y for whatever value of x is provided. Of course, when we write the square function we don't know what the value of x is. The value will be provided when square is invoked.

Whatever value is assigned to y becomes the result for this function.

2.8 Invoking functions

Once the file square.m exists, we can invoke square just like a built-in function.

```
>> square (3)
```

```
ans = 9
>> square (4)
ans = 16
```

Just as with a built-in function, the argument can be a variable.

```
>> bob = 5
>> square (bob)
ans = 25
```

Notice that the name of the variable does not have to be `x`. The name `x` only appears inside the function file; the person invoking the function doesn't know (or care) what the input variable is called inside the function file.

Similarly, if we assign the result of the function to a variable, the name of the variable doesn't have to be `y`.

```
>> fred = square (bob)
fred = 25
```

I deliberately chose the meaningless names `bob` and `fred` to emphasize these points:

1. When you invoke a function, you can use any kind of expression as an argument. If you use a variable, the name of the variable has nothing to do with the name of the input variable inside the function.

2. When you get a result from a function, you can do whatever you want with it. You can display it, send it off to another function, or assign it to a variable. If you assign it to a variable, the name of the variable has nothing to do with the name of the output variable inside the function.

These two rules make it possible to use the functions you define as easily as the built-in functions. For example, we can use `square` to compute the hypotenuse of a triangle using the Pythagorean theorem:

```
>> a = 3
>> b = 4
>> c = sqrt (square(a) + square (b))
c = 5
```

2.9 Local variables

The variables that you use inside a function are **local** to that function, which means that they only exist inside the function and they do not affect variables outside the function, even variables with the same name!

For example, if (for some reason) we change `square` so that it modifies a variable named `a`, that change has no effect outside the function.

Here is the new version of `square`:

```
function y = square (x)
% SQUARE  Returns x raised to the second power.
a = 17
y = x^2
```

And here's what happens when we invoke it.

```
>> square (3)
a = 17
y = 9
ans = 9
>> a
??? Undefined function or variable 'a'.
```

As expected, it assigns a value to `a` and then computes the square of the input variable. But after the function finishes, the local variables disappear. If we try to access them, we get an error.

Conversely, if a variable exists outside a function, you cannot refer to it inside a function. For example, this version of `square` refers to a variable named `b`:

```
function y = square (x)
% SQUARE  Returns x raised to the second power.
y = x^2 + b
```

Even if we create a variable named `b` in the interpreter, the function can't access it:

```
>> b = 5
b = 5
>> square (3)
??? Undefined function or variable 'b'.

Error in ==> square.m
On line 3  ==> y = x^2 + b
```

In this case, MATLAB reports where in the program the error occurred, which is often useful for debugging. On the other hand, keep in mind that MATLAB is not always right!

2.10 Environments

An envionment is a set of variables and their values. When you assign a value to a variable for the first time, you add a new variable to the current environment.

The MATLAB interpreter provides a top-level environment that contains the variables you define interactively. The `who` command lists all the variables in the top-level environment.

```
>> a = 1
a =     1
>> b = 2
b =     2
>> who
Your variables are:

a  b
```

The `clear` command removes variables from the environment.

```
>> clear a
>> who
Your variables are:

b
```

When a function begins execution, its environment contains the input variables and the output variables. As new variables are created, they are added to the environment of the current function.

2.11 Environment diagrams

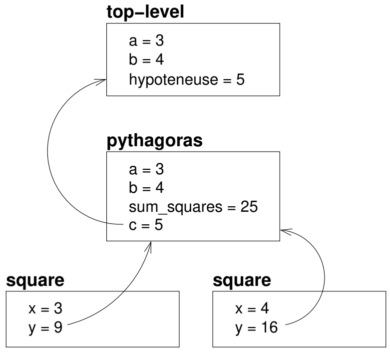
It is often useful to draw a diagram showing a set of functions and the the environment for each. For example, if we write a function called `pythagoras`

```
function c = pythagoras (a, b)
    sum_squares = a^2 + b^2;
    c = sqrt (sum_squares)
```

And we invoke `pythagoras` from the top-level like this

```
>> a = 3
>> b = 4
>> hypotheneuse = pythagoras (a, b)
```

Then we would draw an environment diagram like this



Each environment is represented with a rectangle with the name of the function outside and all the variables inside, with their values.

Notice that since `square` is invoked twice, it appear in the diagram twice, and that the values of the input and output variables are different the second time it is invoked.

Also notice that there can be variables with the same name in different environments. For example, both `pythagoras` and the top-level environment have variables named `a` and `b`. They are not the same variable. If `pythogoras` changed its version of `a`, it would have no effect on the top-level version of `a`.

The arrows in the diagram show the result of each function being returned to the environment that invoked it. When you invoke a function, you pass information to it in the form of arguments and you get information back in the form of the result. Other than that, functions are isolated from each other—each works within its own environment. This isolation makes it possible to design of complex programs with many functions without being overwhelmed by the interactions among the functions.

2.12 Help

The `help` command works for the functions you define as well as for the built-in functions.

```
>> help square
  SQUARE  Computes x raised to the second power.
```

The result is the comment we put in the second line of the program. If your function file includes a series of comments starting on the second line, the help command prints them all.

2.13 Order of evaluation

MATLAB uses the normal evaluation rules of arithmetic. Multiplication and division happen before addition and subtraction. Exponentiation happens before multiplication and division. Operators with the same precedence are evaluated from left to right.

These examples demonstrate the rules.

```
1 + 2 * 3          % multiplication takes precedence
1 * 2 ^ 3          % exponentiation takes precedence
1 * 2 / 3          % evaluate from left to right
```

Of course, you can use parentheses to change the order of evaluation.

```
(1 + 2) * 3        % addition happens first
(1 * 2) ^ 3        % multiplication happens first
1 * (2 / 3)        % division happens first
```

Incidentally, these examples also demonstrate the possibility of putting comments at the end of a line of code.

Chapter 3

Newton

3.1 Root finding

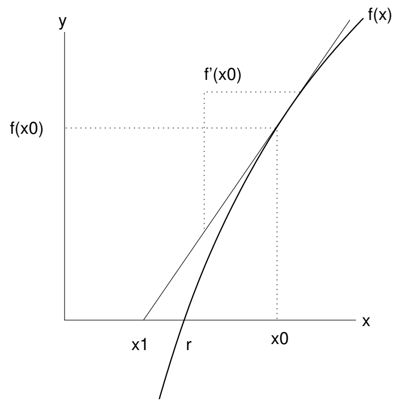
Root finding is the process of finding the places where a function is zero, or (to say the same thing a different way) the values that satisfy an equation. In this section we consider only functions of a single variable, $f(x)$, and try to find the values of x for which $f(x) = 0$. Of course, there might not be any; the function may never cross the x axis. Or there may be an infinite number of roots.

The techniques we use can be extended to equations with multiple variables; for example, we might want to find the values of x, y, and z for which $x^2 + y^2 + z^2 - 1 = 0$. But we won't do that here.

It may not be immediately obvious why root finding is useful (either for one variable or more), but we will look at a number of real-world problems that can be formulated as root-finding problems.

3.2 Newton's method

Newton's method is a root finding technique that we can use for any function that has a derivative. The following figure demonstrates the basic idea.



Suppose that we want to know r, the true root where $f(x)$ crosses the x-axis. If we have an estimate of r, which we call x_0, then we can find a better estimate, which we call x_1, using the function and its derivative.

Assume that we can evaluate $f(x_0)$, which gives us a sense of how far we are from satisfying the equation. Also assume that we can evaluate the derivative, $f'(x)$, which is the slope of the function as it passes through x_0.

Now all we have to make is one more assumption. Unlike the other two, though, this one is only approximately true (which is to say, false). We assume that $f(x)$ is a straight line. If it were, then we could use the value of the function and the slope to compute the exact value of r. It would be:

$$r = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{3.1}$$

Of course, if $f(x)$ were really a linear function we would not be using a numerical method at all; we would just solve it analytically. So if $f(x)$ is nonlinear, the best we can do is compute

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{3.2}$$

and hope that the new estimate, x_1, is better than the old one. If not, then we repeat the process and generate a series of approximations. If all goes well, then this series converges on the true root of the equation. In most cases, this works, and Newton's method converges quickly.

3.3 Newton's square root algorithm

As an example, we can use Newton's method to find the square root of a number, a. If x is the square root of a then the following equation holds:

$$f(x) = x^2 - a = 0 \tag{3.3}$$

Finding the value of x that satisfies this equation is the same as finding the square root of a.

If we start with an initial estimate of the root, x_0, then we can improve the estimate by applying Newton's method:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{x_0^2 - a}{2x_0} = 1/2(x_0 + a/x_0) \tag{3.4}$$

In MATLAB, that looks like this:

```
>> a = 4                          % compute the square root of 4
>> x0 = 3                         % the initial estimate is 3
>> x1 = 1/2 * (x0 + a/x0)
x1 = 2.16666666666667
```

The first step is clearly an improvement, but we are still not very close to the true root (um, 2, right?). We can compute a better estimate from x_1:

```
>> x2 = 1/2 * (x1 + a/x1)
x2 = 2.00641025641026
```

This is getting good.

3.4 Multiple assignment

I haven't mentioned it, but you might have figured out by now that you can make multiple assignments to the same variable. In that case, the second assignment erases the old value of the variable and replaces it with the new variable.

For Newton's method, it might be inconvenient to use different variables for each step in the sequence x_0, x_1, x_2, etc. Instead, we could write the assignment

```
>> x = 1/2 * (x + a/x)
```

This statement is a good reminder that the equals sign in MATLAB is not the same as the equals sign in mathematics. In mathematics, it is a statement of equality—both sides of the equation have the same value. In MATLAB it is an assignment—it evaluates the right side and stores the result in the variable on the left side.

In mathematics, equality is commutative, so if $a = b$ then we also know that $b = a$. The assignment statement is not commutative. Most of the time is it not even legal to switch sides. For example

```
>> 1/2 * (x + a/x) = x
```

is not a legal statement because the left side does not name a variable. Even when an assignment can be reversed, it doesn't *mean* the same thing. `a = b` changes the value of `a`, but `b = a` changes the value of `b`.

I might be making too much of this, but I think that it is a bad idea to use the equals sign for assignment statements. Other languages use `:=` or `<=`. I particularly like the latter because it shows the direction the assignment goes in.

Repeated assignment to the same variable can make programs harder to read, so I don't generally recommend it. But for something like Newton's method, where we are computing a series of approximations, it is convenient:

```
>> x = 3
x = 3
>> x = 1/2 * (x + a/x)
x = 2.16666666666667
>> x = 1/2 * (x + a/x)
x = 2.00641025641026
>> x = 1/2 * (x + a/x)
x = 2.00001024002621
>> x = 1/2 * (x + a/x)
x = 2.00000000002621
>> x = 1/2 * (x + a/x)
x = 2.00000000000000
```

Notice how quickly the series converges on the correct value. An interesting property of Newton's method is that, in many cases, the number of correct digits in the estimate doubles after each improvement.

3.5 Loops

This kind of repetition happens all the time in programs. In fact, automating repetitive processes is one of the things computers do best.

Pretty much every programming language provides a structure called a **loop** that executes instructions repeatedly. In MATLAB one of these structures is called a `for` loop. There are several forms; here is one:

```
>> for j = 1:5
j
end
```

The first line is the head of the loop. It creates a **loop variable** named `j` and it specifies the sequence of values that will be assigned to `j`, in this case the numbers from 1 to 5.

Each time through the loop, MATLAB assigns a new value to `j` and then executes the **body** of the loop, which includes all the statements inside the loop (up to the keyword `end`.

The output of this loop is

```
j = 1
```

```
j = 2
j = 3
j = 4
j = 5
```

In other words, the body of the loop executed 5 times, each time displaying the current value of `j`.

When a for loop appears in a script, it is conventional to indent the statements in the body, to make it clear that they are inside the loop:

```
x = 3
for j = 1:5
    x = 1/2 * (x + a/x)
end
```

The first statement, `x = 3`, is outside the loop, so it only executes once. The body of the loop `x = 1/2 * (x + a/x)` executes 5 times, producing this output:

```
x = 3
x = 2.16666666666667
x = 2.00641025641026
x = 2.00001024002621
x = 2.00000000002621
x = 2.00000000000000
```

3.6 Strings

The values that MATLAB works with come in several flavors, more commonly called `types`. The only type we have seen so far is the floating-point number. The second type is the **character string**, also known simply as the string.

A character is a letter, number or symbol—pretty much anything on the keyboard plus a few. A string is a sequence of characters that are "strung" together.

The MATLAB, strings are enclosed in single-quote marks, as in

```
>> t = 'Hello'
t = Hello
```

Notice that when MATLAB displays a string it omits the quotation marks. That can be confusing, because it makes it more difficult to see the difference between the number 123 and the string `'123'`. But there is a difference! For example, if `x` is the number 123, then we can add one:

```
>> x = 123
x = 123
>> x + 1
ans = 124
```

```
%
and get the answer we expect.  But if {\tt s} is the string {\tt 123},
and we try to add one,

\begin{verbatim}
>> s = '123'
s = 123
>> s + 1
ans = 50    51    52
```

we get something screwy! Later on I'll explain why `'123' + 1` is `50 51 52`, but for now the moral is

> Keep track of what is a number and what is a string, and don't perform arithmetic on strings.

Actually, there is one operation you might want to perform on strings. You can join them end to end by enclosing them in square brackets.

```
>> s = 'Hello, '
>> t = 'World!'
>> [s t]
ans = Hello, World!
```

This operation is called **concatenation**. You can combine any number of string expressions in brackets:

```
>> [s 'Cruel ' t]
ans = Hello, Cruel World!
```

Notice that it is legal, and often useful, to include spaces in a string. A space is a character just like any other.

3.7 Displaying values

So far we have been taking advantage of MATLAB's tendency to display the value of a variable whenever it is assigned. This behavior is useful for debugging, but sometimes overwhelming. We can suppress it by adding a semi-colon at the end of the statement. Looking again at Newton's square root program:

```
x = 3;
for j = 1:5
    x = 1/2 * (x + a/x);
end
x
```

This version does not display the initial value of x or any of the intermediate values. Only at the end, after exiting the loop, it displays the final value.

If you just want the value of a variable, without the name, you can use the disp command:

```
>> x = 2
x = 2
>> disp (x)
    2
```

If you want even more control of the output format, you can use the fprintf command. The first "f" in fprintf stands for "file," because this command is also used to put data into files. The second "f" stands for "format." The word "print" in this context means, "display the result on the screen," and has nothing to do with the usual kind of printing (on paper).

The first argument of fprintf is the **format string**, which, in the simplest case, is just a plain old string of characters:

```
>> fprintf ('Hello')
Hello
```

If the characters %f appear anywhere in the format string, they mean, "take the next argument and display it as a floating-point number."

```
>> fprintf ('The value of pi is approximately %f', pi)
The value of pi is approximately 3.141593
```

By default, the floating-point format displays 6 digits after the decimal point. The %f sequence can appear more than once, but there must be enough additional arguments to match each one.

```
>> fprintf ('pi is %f and e is %f', pi, exp(1))
pi is 3.141593 and e is 2.718282
```

If you call fprintf repeatedly, all the output appears on one line, which can be useful. To separate lines, you can put the linefeed sequence,
n, in the string. For example, here is yet another version of the square root algorithm:

```
for j = 1:5
    x = 1/2 * (x + a/x);
    fprintf ('j = %f   x = %f\n', j, x)
end
```

And here is the output

```
j = 1.000000   x = 2.500000
j = 2.000000   x = 2.050000
j = 3.000000   x = 2.000610
j = 4.000000   x = 2.000000
j = 5.000000   x = 2.000000
```

As an exercise, take this code and encapsulate it in a function called `newton` that takes a single argument `a` and that returns the square root of `a` (or at least the approximation you get after five iterations). You can use the value of `a` as the initial estimate.

3.8 Helper functions

The first function in an m-file is the top-level function. It should have the same name as the file, and it can be invoked from the MATLAB interpreter or from any other m-file.

You can also put other functions in the same file. They can be invoked by the top-level function or by each other, but they are not visible to the interpreter or other m-files. These functions are sometimes called **helpers** because they help the top-level function. In the following example, we use two helper functions to implement Newton's method for something more complicated than square roots.

3.9 Scary numbers

Let's assume that you have two kids some time in the next 10 years, and that you are planning to send them to college[1]. Also, assume that by the time they go to college tuitions have doubled, to $60,000 per year. That means you have about 25 years to pull together $480,000.

If you start saving now by depositing money in a savings account every month, the total amount you will have in 25 years is

$$a = \frac{p}{i}[(i+1)^n - 1] \tag{3.5}$$

where a is the balance in the account, p is the amount of the monthly deposit, n is the number of months you keep this up (25 years $\times$ 12 months per year $=$ 300), and i is the interest rate per period. Assuming you can afford to deposit $100 per month, what is the minimum monthly interest rate you need in order to make it?

[1]This section presents a problem adapted from Faires and Burden's *Numerical Methods*, problem 2.4.15.

3.10 Formulating the problem

The first step to solving this problem is to formulate it as a root-finding problem. We would like to find the value of i that makes Equation 3.5 true for the given values of p, n, and a. In other words, we want to find a root of the function

$$f(i) = \frac{p}{i}[(i+1)^n - 1] - a \tag{3.6}$$

A bit of mathematical vocabulary will help us talk about functions like this. In this function, i is a variable; we are interested in how $f(i)$ varies with i. p, n and a are not variables in the technical sense; they are parameters (constants that control the behavior of the function).

Another way to think of this distinction is that the parameters might vary from problem to problem, but only the variable will be changing during this problem.

The reason I raise (and possibly belabor) this point is that computer scientists and mathematicians use the word "variable" differently. In computer science a variable is a named location that stores a value, and it doesn't really matter how it is used.

So, in the MATLAB program `i`, `p`, `n` and `a` are all variables.

3.11 The helper functions

A good way to write programs like this is to identify small functions that are likely to be useful, then write and debug those functions one at a time.

In this case we are planning to use Newton's method, so we know we have to evaluate the function and its derivative. The function looks like this:

```
function f = func (a, p, n, i)
    f = p/i * ((i+1)^n - 1) - a;
```

Notice the semi-colon I threw in so that this function doesn't display anything when it is invoked. Its only effect is to return a value.

To test the function, I created a new file named `interest.m` with a top-level function called `interest`. When we are done, `interest` will be the function that solves the problem and returns the answer, but for now we are going to use it to test the helper functions.

Here is what the first version of `interest` looks like:

```
function result = interest ()

    a = 480000;               % how much money we need
    p = 100;                  % monthly deposit
    n = 12 * 25;              % how many months we have
    i = 0.01;                 % a monthly interest rate for testing
```

```
    result = func (a, p, n, i);
```

The result when I invoke `interest` is

```
>> interest
ans = -2.921153373807556e+05
```

which means that at an interest rate of 1% per month, I am going to be $292,115 short. That gives us a rough idea of where we stand.

Of course, when I wrote this it worked the first time. But if it didn't, then the first thing I would have done is remove the semi-colon so I could see what value of $f(i)$ was being computed. My next step would be to break the complex expression in `func` into several assignments to temporary variables.

Assuming we get `func` to work, the next step is to evaluate the derivative. Written in mathematical notation, it's

$$\frac{df}{di}(i) = \frac{p}{i^2}[(i+1)^n - 1] - \frac{p}{i}n(i+1)^{n-1} \tag{3.7}$$

As an exercise, translate that into MATLAB.

3.12 Newton himself

Once the helper function are working, we have to assemble them into a complete program. The first step is to fill out the top-level function, `interest`. It should take `p`, `n` and `a` as arguments and return the estimate of `i` that it finds. The initial estimate is `i = 0.01`, which is already pretty optimistic—you don't find many savings accounts paying 1% per month.

```
function result = interest (a, p, n)
    % result = interest (a, p)
    % computes the minimum interest needed to reach the amount A
    % after n periods, given a periodic payment of P
    i = 0.01;

    for j = 1:10
        fprintf ('%f   %f\n', j, i);

        f = func (a, p, n, i);
        fp = deriv (a, p, n, i);

        i = i - f / fp;
    end

    result = i
```

The loop iterates 10 times. Each time through the body, it displays the loop variable and the current estimate of i. Then it computer $f(i)$, $f'(i)$ and the improved estimate of i.

The output variable is `result`. As usual, the last line of the function is an assignment to the output variable.

Putting it all together, we can invoke `interest` with the parameters in the problem:

```
>> interest (480000, 100, 25*12)
1.000000   0.010000
2.000000   0.017305
3.000000   0.015153
4.000000   0.014326
5.000000   0.014229
6.000000   0.014228
7.000000   0.014228
8.000000   0.014228
9.000000   0.014228
10.000000   0.014228

ans =

   0.01422819647289
```

This is not good news! In order to put your kids through school, you have to find an investment that pays more than 1.4% interest per month, or else you are going to have to come up with more than $100 per month.

3.13 Divide, conquer and glue

This example demonstrates one of the most common and powerful problem-solving ideas in computer science, **divide, conquer and glue**. "Divide" refers to the process of taking a large problem and breaking it into smaller pieces. "Conquer" refers to the process of solving the pieces. "Glue" refers to the process of assembling the solved pieces into a solution to the original problem.

In this case the helper functions are the small solutions and the top-level function is the glue that assembles the pieces and provides a general interface for solving an entire category of problems.

By the way, there is a name for a mechanical process, like a program, that solves a general category of problems. It is called an **algorithm**. Newton's method is an algorithm for approximating a root of a nonlinear function.

Chapter 4

Conditionals, graphics and recursion

4.1 Conditional execution

In order to write useful programs, we almost always need the ability to check certain conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if x > 0
    disp ('x is positive');
end
```

The expression after `if` is called the condition. If it is true, then the indented statements get executed. If the condition is not true, nothing happens.

The condition can contain any of the **comparison operators**:

```
    x == y              // x equals y
    x ~= y              // x is not equal to y
    x > y               // x is greater than y
    x < y               // x is less than y
    x >= y              // x is greater than or equal to y
    x <= y              // x is less than or equal to y
```

Although these operations are probably familiar to you, the syntax MATLAB uses is different from mathematical symbols like $=$, $\neq$ and $\leq$. A common error is to use a single `=` instead of a double `==`. Remember that `=` is the assignment operator, and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

4.2 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
if x < 0
    disp ('x is negative');
else
    disp ('x is non-negative');
end
```

If `x` is negative, the first branch of the conditional is executed. to that effect. If the condition is false, the second branch is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

4.3 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if`s and `else`s:

```
if x > 0
  disp ('x is positive')
elseif x < 0
  disp ('x is negative')
else
  disp ('x is zero')
end
```

Again, exactly one of these branches will be executed.

You can string together as many `elseif`s as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements indented properly, you are less likely to make syntax errors and you can find them more quickly if you do.

Also, be sure to use `elseif` for chained conditionals. If you write `else if`, you are writing a nested conditional.

4.4 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if x == 0
    disp ('x is zero');
else
```

```
        if x > 0
            disp ('x is positive')
        else
            disp ('x is negative')
        end
    end
```

There is now an outer conditional that contains two branches. The first branch contains a simple `disp` statement, but the second branch contains another conditional statement, which has two branches of its own. Fortunately, those two branches are both `disp` statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

4.5 The return statement

The `return` statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

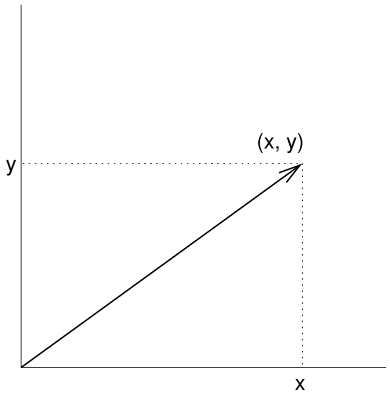
```
function printLogarithm (x)
    if x <= 0
        disp ('Positive numbers only, please.');
        return;
    end

    fprintf ('The logarithm of x is %f.\n', log (x));
```

This defines a function named `printLogarithm` that takes an argument named `x`. Notice that there is no return variable. This function will display output but produce no return value.

The first thing it does is check whether `x` is less than or equal to zero, in which case it prints an error message and then uses `return` to exit the function. The remaining lines of the function are not executed.

4.6 Vectors

In a math class you have probably seen vectors used to represent points in two- or three-dimensional space. A 2-D vector is usually written (x, y) and it representes the point in the plane that is x units from the origin along the x-axis and y units from the origin along the y-axis.



Vectors are exactly the sort of thing MATLAB was designed to work with, so it provides lots of built-in functions for vectors. To create a new vector, just put the coordinates in square brackets.

```
>> [3, 4]
ans = 3  4
```

When MATLAB displays a vector it puts spaces between the coordinates but it doesn't print the brackets.

You can assign vectors to variables.

```
>> v = [3, 4]
v = 3  4
```

You can extract the coodinates from a vector using the parentheses operator:

```
>> v(1)
ans = 3
>> v(2)
ans = 4
```

The number in parentheses is the **index** that indicates which coordinate you want. A two-dimensional vector has two coordinates, numbered 1 and 2. If you use any other index, you get a run-time error.

You can pass vectors as arguments to functions.

```
>> norm (v)
ans = 5
```

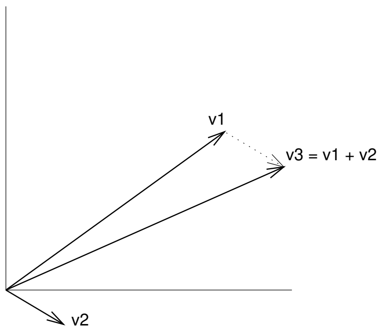
The built-in function `norm` computes the length of the vector, which is $\sqrt{x^2 + y^2}$. You can also write a function that returns a vector as a return value:

```
function vn = normalize (v)
    vn = v / norm (v);
```

Notice that it is legal to divide a vector by a scalar (a scalar is a single number that is not a vector). MATLAB understands that we want to divide through, that is, divide each coordinate by the scalar. In this case, dividing through by the length of a vector yields a vector with length 1 (and the same direction as the original).

```
>> vn = normalize (v)
vn =  0.6000    0.8000

>> norm (vn)
ans =  1
```

The operators + and - work on vectors; they perform vector addition and subtraction. The following figure demonstrates vector addition.



We can vizualize the sum of v1 and v2 by imagining a copy of v2 with its tail at the head of v1 (shown as a dotted line in the figure).

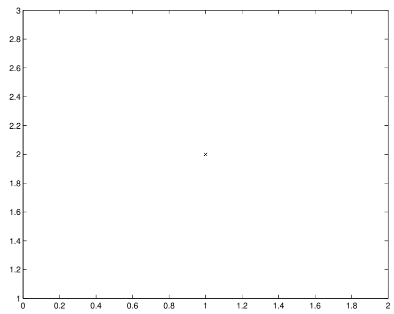
4.7 Plotting

MATLAB provides a variety of built-in functions for generating plots. The plot command generates a simple plot:

```
>> plot (1, 2, 'x')
```

This command creates a new figure and plots a single point, at the coordinates $(1, 2)$. The third argument is a string that indicates how the point should be represented. To see the other options, try help plot.

MATLAB chooses the axes for the figure automatically. The result is



To plot a line we will use a built-in command called `line`. Unfortunately, there is no straightforward way to draw lines using vectors. Instead, I have written a helper function called `draw_line` that takes two vectors as arguments and that draws the line between them.

```
function draw_line (v1, v2)
    z = [v1;v2];
    line (z(:,1), z(:,2))
```

In this function, `z` is a matrix. We haven't learned about matrices yet, so this function won't make sense for another few chapters.

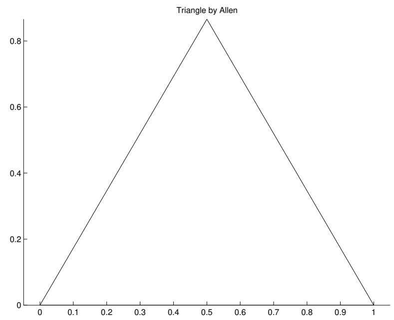
The following example uses `draw_line` to draw a triangle.

```
function triangle ()
    clf; axis equal
    title 'Triangle by Allen'

    v1 = [0,0];
    v2 = [0.5, sqrt(3)/2];
    v3 = [1,0];

    draw_line (v1, v2)
    draw_line (v2, v3)
    draw_line (v3, v1)
```

The points `v1`, `v2` and `v3` are the vertices of an equilateral triangle. The base is 1 unit long, so the height is $\sqrt{3}/2$.

The command `clf` stand for "clear figure"; it clears the figure window before drawing new lines. The command `axis equal` instructs MATLAB to choose axes so that the x- and y-directions are scaled equally. That way the triangle appears, appropriately, equilateral. You can put more than one command on a line, as long as you put semi-colons between them.

The command `title` puts a title on the figure, which is useful for identifying figures when they are printed. Here is the result of this function.



4.8 Recursion

We have seen several examples where one function invokes another. It is also legal for a function to invoke itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, look at the following function:

```
function countdown (n)
    if n == 0
      disp ('Blastoff!')
    else
      disp (n)
      countdown (n-1)
    end
```

The name of the function is `countdown` and it takes an argument named `n`. If the argument is zero, it prints the word "Blastoff." Otherwise, it prints the number and then invokes a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we invoke this function like this?

```
>> countdown (3)
```

The execution of `countdown` begins with `n=3`, and since `n` is not zero, it prints the value 3, and then invokes itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not zero, it prints the value 2, and then invokes itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not zero, it prints the value 1, and then invokes itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is zero, it prints the word "Blastoff!" and then returns.

The countdown that got `n=1` returns.

The countdown that got `n=2` returns.

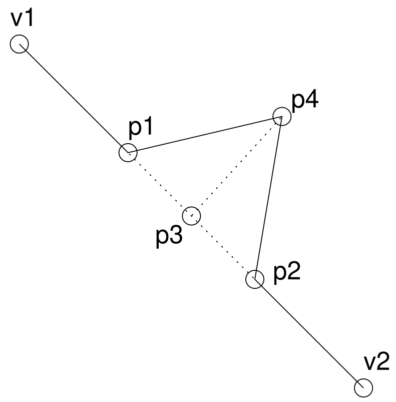
The countdown that got `n=3` returns.
And then you're done. So the total output looks like:

```
    3
    2
    1
Blastoff!
```

The process of a function invoking itself is called **recursion**, and such functions are said to be **recursive**. As an exercise, draw an environment diagram for this sequence of function invocations.

4.9 More vectors

Getting back to the triangle program, we can make it more interesting by writing a function that draws crooked lines. The arguments are the same as for `draw_line`, but instead of drawing a single line from `v1` to `v2`, it draws four lines, as shown in the figure:



The points `p1`, `p2`, `p3` and `p4` refer to variables in the following function:

```
function draw_crooked (v1, v2)
    v = v2 - v1;

    p1  = v1 + v / 3;
    p2  = v1 + 2 * v / 3;
    p3  = v1 + v / 2;

    length = norm (v);
    height = length/3 * sqrt(3)/2;
```

```
    p4 = p3 + height * normalize (perpendicular (v));

    draw_line (v1, p1);
    draw_line (p2, v2);
    draw_line (p1, p4);
    draw_line (p4, p2);
```

This is the most complicated function we have seen so far, so let's look at it in detail. First we compute the difference between the end points, `v`, which is a vector that has the direction and length of the line between `v1` and `v2`.

We can use `v` to compute `p1`, which is the sum of `v1` and a vector that is in the same direction as `v2`, but only $1/3$ as long. Similarly, `p2` is in the same direction, but twice as far from `v1`.

`p3` is easy to compute; it's halfway between `v1` and `v2`. We could have written `p3 = (v1 + v2) / 2`. Same thing.

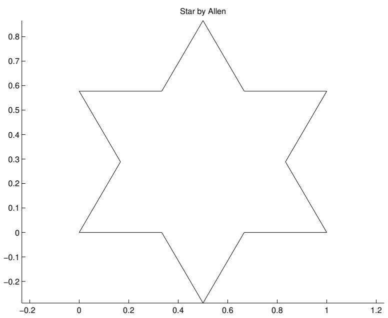
Computing `p4` is harder. We need to find the vector that is perpendicular to `v`, with length that is one third of $\sqrt{3}/2$ times the length of `v`. Well, the length of `v` is `length`, computed with the `norm` function we have already seen. So the height of the small triangle is `length/3 * sqrt(3)/2`.

To find the perpendicular vector, we use `perpendicular`, defined like this:

```
function vp = perpendicular (v)
    vp = [-v(2), v(1)];
```

If you have taken linear algebra, you might remember that in two dimensions you can find a perpendicular vector by reversing the coordinates and negating one of them.

If we modify `triangle` so that it invokes `draw_crooked` instead of `draw_line` we get the following:



Notice that MATLAB automatically adjusts the axes so the lines fit in the figure.

4.10 Fractals

In 1988 Benoit Mandelbrot published a book called `The Fractal Geometry of Nature` that made the best-sellers list, a rare feat for a math book. It is full of pretty pictures of fractals, which are geometric figures that are self-similar; that is, each part of the figure looks like a scaled down version of the whole.

These kinds of shapes appear frequently in nature, which may be why we find them beautiful. The florets that make up a head of cauliflower, for example, look like small heads of cauliflower. Many other plants are self-similar in this way. If you draw a map of a riverbank or coastline, it tends to look the same regardless of the scale, from inches up to hundreds of miles. Clouds and mountain ridges have the same property.

There is a natural connection between fractals and recursion. The easiest way to generate a fractal is to write a recursive function. In fact, it is trivial to take `draw_crooked` and make it recursive. All we have to do is replace the invocation of `draw_line` with a recursive call:

```
function draw_crooked (v1, v2)
    v = v2 - v1;

    p1  = v1 + v / 3;
    p2  = v1 + 2 * v / 3;
    p3  = v1 + v / 2;

    length = norm (v);
    height = length/3 * sqrt(3)/2;
    p4 = p3 + height * normalize (perp (v));

    draw_crooked (v1, p1);
    draw_crooked (p2, v2);
    draw_crooked (p1, p4);
    draw_crooked (p4, p2);
```

When `draw_crooked` is invoked, it computes the same points it computed before, but instead of drawing a straight line between each pair of points, it draws a crooked line. How does it draw the crooked line? By invoking itself!

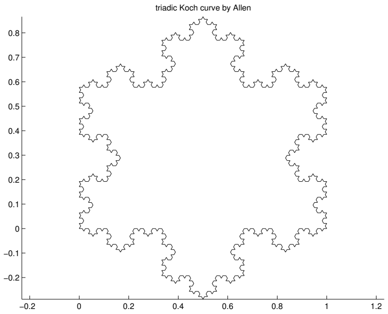
But wait a minute. If the only way to draw a crooked line is by drawing four crooked lines, then how can we ever stop? If we run `draw_crooked` the way it is, it will run forever. This situation is called an **infinite recursion** and it is generally not a good idea. In MATLAB the function can recurse 100 times, and then it will cause a run-time error.

To fix this problem, we can borrow a trick from `countdown` and add a third parameter, `n`. Each time we recurse, we decrease `n` by one. When it gets to 0, we stop the recursion by drawing a simple line and invoking `return`. Remember that `return` terminates the function immediately. Since the rest of the function is not executed, we don't reach the recursive calls, and the recursion stops.

```
function draw_snow (v1, v2, n)
    if (n == 0)
        drawline (v1, v2);
```

```
    return
end

v = v2 - v1;
p1  = v1 + v / 3;
p2  = v1 + 2 * v / 3;
p3  = v1 + v / 2;

length = norm (v);
height = length/3 * sqrt(3)/2;
p4 = p3 + height * normalize (perp (v));

draw_snow (v1, p1, n-1);
draw_snow (p2, v2, n-1);
draw_snow (p1, p4, n-1);
draw_snow (p4, p2, n-1);
```

I changed the name of the function to `draw_snow` because if you invoke it with `n=4`, you get this:



To the untrained eye, this might look like a snowflake, but it is actually the triadic Koch curve.

4.11 Convention and divine law

In several places I have used the phrase "by convention" to indicate design decisions that are arbitrary in the sense that there are no significant reasons to do things one way or another, but dictated by convention.

In these cases, it is a good idea to be familiar with convention and use it; among other things, it makes your programs easier for others to understand. At the same time, it is important to distinguish between (at least) three kinds of rules:

Divine law: This is my phrase to indicate a rule that is true because of some underlying principle of logic or mathematics, and that is true in any programming language (or other formal system). For example, the technique we used to rotate a vector to find a perpendicular comes from linear algebra; it has nothing to do with MATLAB.

Rules of MATLAB: These are the syntactic and semantic rules of MATLAB that you cannot violate, because the resulting program will not compile or run. Some are arbitrary; for example, the fact that we use square brackets to concatenate strings. Others reflect underlying limitations of the interpreter or the process of execution. For example, you cannot invoke helper functions from outside their files.

Style and convention: There are a lot of rules that are not enforced by the compiler, but that are essential for writing programs that are correct, that you can debug and modify, and that others can read. Examples include indentation of function bodies and `if` statements, as well as conventions for naming variables and functions.

As we go along, I will try to indicate which category various things fall into, but you might want to give it some thought from time to time.