Chapter 4

Mandelbrot

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. 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 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. In this case the two branches are `disp` statements, but 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:

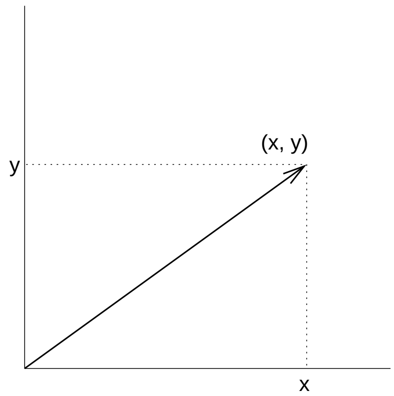
```
function printLogarithm (x)
    if x <= 0
        disp ('Positive numbers only, please.')
        return
    end

    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 displays output but produces 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 uses `return` to exit the function. In this case, 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. Try it.

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 $\sqrt{x^2 + y^2}$.

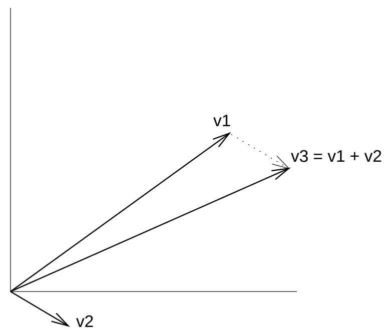
You can also write a function that returns a vector as a return value:

```
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).

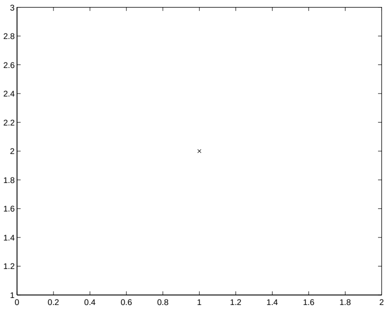
4.7 Plotting

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 subfunction 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))
end
```

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.

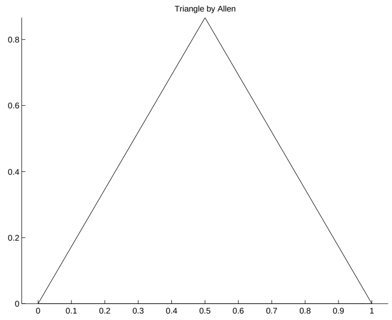
The following example uses `draw_line` to draw a triangle.

```
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)
end
```

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` stands 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
end
```

The name of the function is `countdown` and it takes an input variable named `n`. If `n` is zero, the function prints "Blastoff." Otherwise, it prints `n` 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.

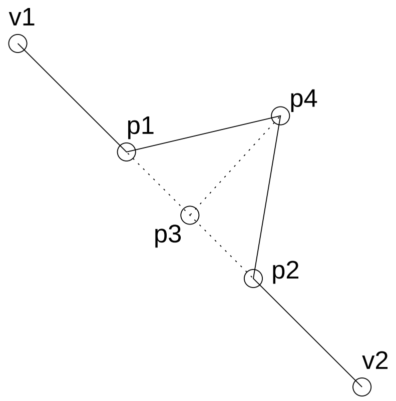
The countdown that got `n=3` returns.
And then you're done. So the total output looks like:

```
    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 workspace 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.

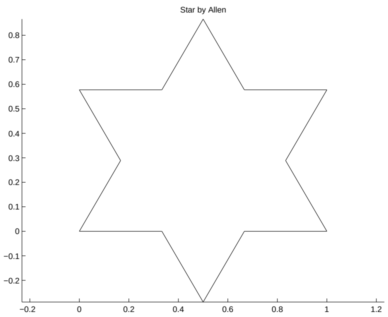
Computing `p4` is harder. We need to find the vector that is perpendicular to `v`, with length that is one third of $\sqrt{3}/2$ times the length of `v`. Well, the length of `v` is `length`, computed with the `norm` function we have already seen. So the height of the small triangle is `length/3 * sqrt(3)/2`.

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!

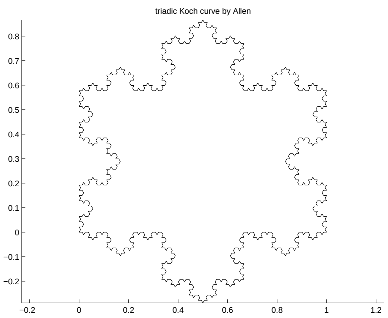
But wait a minute. If the only way to draw a crooked line is by drawing four crooked lines, then how can we ever stop? If we run `draw_crooked` the way it is, it will run forever. This situation is called an **infinite recursion** and it is generally not a good idea. In MATLAB the function can recurse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