Chapter 3

Newton

3.1 Root finding

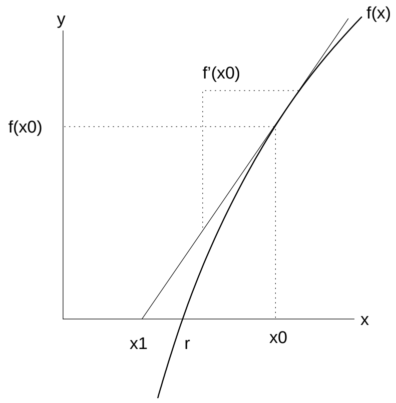
Root finding is the process of finding the places where a function is zero, or (to say the same thing a different way) the values that satisfy an equation. In this section we consider only functions of a single variable, $f(x)$, and try to find the values of x for which $f(x) = 0$. Of course, there might 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, 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

The kind of repetition in the previous example 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 executes 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 a 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 Subfunctions

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. They are called **subfunctions**. In the following example, we use two subfunctions 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?

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

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

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 subfunctions

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 subfunctions.

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 subfunctions 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 subfunctions 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.