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 approximation 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 cannot 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 is 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 9999.5, then the 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 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.

The ellipsis (`...`) allows you to break a big expression or statement 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.

They 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 value you provide (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 hypoteneuse 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 Workspaces

An workspace 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 workspace.

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

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

a  b
```

The `clear` command removes variables from the workspace.

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

b
```

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

2.11 Workspace diagrams

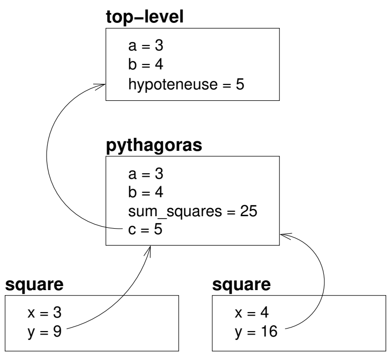
It is often useful to draw a diagram showing a set of functions and the the workspace for each. For example, if we write a function called `pythagoras`

```
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 workspace diagram like this



Each workspace 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 appears 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 workspaces. For example, both `pythagoras` and the top-level workspace 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 workspace 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 workspace. This isolation makes it possible to design 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.