Author Sebastian Coope

When testing maths functions you are expected to test by giving the function and input and comparing the output with an expected value. For simple integer value functions this is relatively easy. For example a factorial function, int fact(int), would take in the value of 5 and return 5 x 4 x 3 x 2 x 1 = 120.

The problem gets a lot more difficult when testing numbers with real arguments.

Real representation

One of the problems is the issue of number representation. For example if you wish to represent the number 0.2 in binary, even though in decimal it has only 2 decimal place, in binary this number is an infinite recurring fraction. This means it is impossible for it to be represented to complete accuracy. Other numbers such as PI cannot be represented perfectly. This makes certain tests hard to do, such as doing the SIN function of PI which would be expected to return 0.

Converting from decimal to binary

We can convert from decimal to binary by multiplying the number by two each time, if the integer part of the result is odd the next bit is a one, otherwise it is a zero.

Here is an example.

0.1

0.1 x 2 = 0.2	Even	Next bit = 0
0.2 x 4 = 0.4	Even	Next bit = 0
0.4 x 2 = 0.8	Even	Next bit = 0
0.8 x 2 = 1.6	Odd	Next bit = 1
1.6 x 2 = 3.2	Odd	Next bit = 1
3.2 x 2 = 6.4	Even	Next bit = 0
6.4 x 2 = 12.8	Even	Next bit = 0
12.8 x 2 = 25.6	Odd	Next bit = 1

This process carries on until the decimal part of the fraction = 0, but in this case (0.1) we can see this is never going to happen, so 0.1 is a recurring binary fraction.

So 0.1 = 0.0001100110011001100110011 going on for ever.

If we try and store this in 8-bits we will lose some precision.

0.00011001

To make is as accurate as possible, if the next bit is a one, we round the last bit up, so in this case we add

0.00000001

So the result is

0.00011010

We can see now that what we store is 1/16 + 1/32 + 1/128 = 0.1015625

The error will be 0.1015625 – 0.1 = 0.0015625

Which as a percentage of 0.1 = 1.5625%

If on the other hand we use floating point (instead of fixed point), the number will be stored as (this is called normalized format)

1.10011001 x 2^--4

Again rounding up the mantissa we get

1.10011010 x 2^--4 = 0.10009765625

Error = 0.0009765625 as a % = 0.9765625%

We can see that using floating point we increase the accuracy of the result.

So we can see that the accuracy a number can be stored in is determined:

By the number value, 0.5 is easier to store in binary than 0.1.

The base used to store the number 0.1 can be stored perfectly in decimal but will recur in binary.

The number of bits in the mantissa.

ULP Unit of least precision

Sometimes defines as the distance between 2 straddling floating point numbers? In practise this means, the maximum rounding error when storing a number is 0.5 ULP

The size of the error when representing fractions is proportional to the size of the number stored. The bigger the number stored the bigger the ULP.

The condition number

The condition number defines the amount of error in the output of a function given an error in the input to the function. A function with very large condition number will be difficult to test to a high degree of accuracy since small errors in the input will lead to large errors in the output.

It is important to note that a well-conditioned function can have poor conditioning overall due to errors in a step in the function.

So for example:

A=45/(x-5)

Y=A*(x-5)

This should simply result in Y=45 however when X is close to 5 the error on the output will grow, this is due to

1) Step 1 is poorly conditioned near x = 5
2) When large values are stored in A this results in rounding errors due to the very large values being stored

Test tolerance

The test tolerance is the largest amount the actual output of the test can differ from the expected output before the test has failed. The test tolerance is calculated using the condition number of the function and the ULP for the values given. This is to make sure the test does not fail due to expected errors in representation and function conditioning.

Ways to test functions

Golden values

Certain values for certain functions are known mathematically, these are often called golden values as they are known to perfect precision.

For example

Log(1) = 0

Log(100)base10=2

Sin(PI/2 radians)=1

Sin(PI radians)=0

Identity tests

Some functions when used together or functions used against their inverse will produce and identity.

For example sin(x)^2 + cos(x)^ 2 = 1

This holds true for whatever x is.

square_root(x) x square_root(x) = x

Testing against an already validated library.

If there is a set of library functions available you can use these to generate test tables to use to perform the tests.

Here are some examples of tests

Function	Input	Output	
Sin	0	0	Golden
Cos	0	1	Golden
Sin	PI/2	1	Golden
Log	1	0	
Log	0	Exception	
Log	100	2	
Log	1000	3	
Log	10000	4	
Log	0.1	-1	
Log	-1	Error exception	
Power(10,0)	1	Golden value	