Circle 1

Falling into the Floating Point Trap

Once we had crossed the Acheron, we arrived in the first Circle, home of the virtuous pagans. These are people who live in ignorance of the Floating Point Gods. These pagans expect

```
.1 == .3 / 3
```

to be true.

The virtuous pagans will also expect

```
seq(0, 1, by=.1) == .3
```

to have exactly one value that is true.

But *you* should not expect something like:

```
unique(c(.3, .4 - .1, .5 - .2, .6 - .3, .7 - .4))
```

to have length one.

I wrote my first program in the late stone age. The task was to program the quadratic equation. Late stone age means the medium of expression was punchcards. There is no backspace on a punchcard machine—once the holes are there, there's no filling them back in again. So a typo at the end of a line means that you have to throw the card out and start the line all over again. A procedure with which I became all too familiar.

Joy ensued at the end of the long ordeal of acquiring a pack of properly punched cards. Short-lived joy. The next step was to put the stack of cards into an in-basket monitored by the computer operator. Some hours later the (large) paper output from the job would be in a pigeonhole. There was of course an error in the program. After another struggle with the punchcard machine (relatively brief this time), the card deck was back in the in-basket.

It didn't take many iterations before I realized that it only ever told me about the *first* error it came to. Finally on the third day, the output featured no messages about errors. There was an answer—a *wrong* answer. It was a simple quadratic equation, and the answer was clearly 2 and 3. The program said it was 1.999997 and 3.000001. All those hours of misery and it can't even get the right answer.

I can write an R function for the quadratic formula somewhat quicker.

```
> quadratic.formula
function (a, b, c)
{
        rad <- b^2 - 4 * a * c
        if(is.complex(rad) || all(rad >= 0)) {
                rad <- sqrt(rad)
        } else {
                rad <- sqrt(as.complex(rad))
        }
        cbind(-b - rad, -b + rad) / (2 * a)
}
> quadratic.formula(1, -5, 6)
     [,1] [,2]
[1,]    2    3
> quadratic.formula(1, c(-5, 1), 6)
                [,1]           [,2]
[1,]  2.0+0.000000i  3.0+0.000000i
[2,] -0.5-2.397916i -0.5+2.397916i
```

It is more general than that old program, and more to the point it gets the right answer of 2 and 3. Except that it doesn't. R merely prints so that most numerical error is invisible. We can see how wrong it actually is by subtracting the right answer:

```
> quadratic.formula(1, -5, 6) - c(2, 3)
     [,1] [,2]
[1,]    0    0
```

Well okay, it gets the right answer in this case. But there *is* error if we change the problem a little:

```
> quadratic.formula(1/3, -5/3, 6/3)
     [,1] [,2]
[1,]    2    3
> print(quadratic.formula(1/3, -5/3, 6/3), digits=16)
[1,] 1.999999999999999 3.000000000000001
> quadratic.formula(1/3, -5/3, 6/3) - c(2, 3)
               [,1]         [,2]
[1,] -8.881784e-16 1.332268e-15
```

That R prints answers nicely is a blessing. And a curse. R is good enough at hiding numerical error that it is easy to forget that it is there. Don't forget.

Whenever floating point operations are done—even simple ones, you should assume that there will be numerical error. If by chance there is no error, regard that as a happy accident—not your due. You can use the `all.equal` function instead of `'=='` to test equality of floating point numbers.

If you have a case where the numbers are logically integer but they have been computed, then use `round` to make sure they really are integers.

Do not confuse numerical error with an error. An error is when a computation is wrongly performed. Numerical error is when there is visible noise resulting from the finite representation of numbers. It is numerical error—not an error—when one-third is represented as 33%.

We've seen another aspect of virtuous pagan beliefs—what is printed is all that there is.

```
> 7/13 - 3/31
[1] 0.4416873
```

R prints—by default—a handy abbreviation, not all that it knows about numbers:

```
> print(7/13 - 3/31, digits=16)
[1] 0.4416873449131513
```

Many summary functions are even more restrictive in what they print:

```
> summary(7/13 - 3/31)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.4417  0.4417  0.4417  0.4417  0.4417  0.4417
```

Numerical error from finite arithmetic can not only fuzz the answer, it can fuzz the question. In mathematics the rank of a matrix is some specific integer. In computing, the rank of a matrix is a vague concept. Since eigenvalues need not be clearly zero or clearly nonzero, the rank need not be a definite number.

We descended to the edge of the first Circle where Minos stands guard, gnashing his teeth. The number of times he wraps his tail around himself marks the level of the sinner before him.