Tip 81: Deprecate floats

Ben Klemens

12 March 2012

level: basic numerics
purpose: Ignore lots of caveats

I don't know if you've noticed, but I stopped using `float`.

There's a lot of advice about how you've got to be careful about avoiding floating-point tricks all the way along. Much of it is still valid today, but much of it is easy to handle quickly: use `double` instead of `float`.

For example, have a look at the caveat on p 24 of *Writing Scientific Software*[1], which advises users to avoid what they call the single-pass method of calculating variances.

They give an example which is ill-conditioned. I reprint their list of numbers below, and you can see that even though the numbers are in the tens of thousands, they differ mostly after the decimal. The authors get terrible results, with a variance that seems off by two orders of magnitude.

Apophenia uses the advised-against single-pass method, as does the GSL. How bad are the results? Not bad at all, actually.

Here's the code to run their example. I do the example twice: once with the ill-conditioned version, and once after subtracting 34,120 from every number, which thus gives us something that even a plain `float` can handle with full precision. We can be confident that the results given the not-ill-conditioned numbers are accurate.

```
#include <apop.h>

int main(){
    apop_data *d = apop_data_fill(apop_data_alloc(1, 6),
           34124.75,
           34124.48,
           34124.90,
           34125.31,
           34125.05,
           34124.98);

    double m, var;
    apop_matrix_mean_and_var(d->matrix, &m, &var);
```

[1] `http://books.google.com/books?id=E6a8oZOS8noC`

```
    printf("mean: %.10g var: %.10g\n", m, var*6/5.);

    apop_data_fill(d,
           4.75,
           4.48,
           4.90,
           5.31,
           5.05,
           4.98);

    apop_matrix_mean_and_var(d->matrix, &m, &var);
    printf("mean: %.10g var: %.10g\n", m, var*6/5.);
}
```

- Apophenia returns the population variance; we scale to produce the sample variance, which the authors prefer.
- I used `%g` as the format specifier in the `printf`s; that's the 'general' form, which accepts both floats and doubles.
- Internally, `apop_matrix_mean_and_var` uses a `long double`, following the basic principle that you should keep your intermediate values one step more precise to prevent intermediate roundoffs from aggregating into problems. It used to just use a `double`, and the results weren't actually different.

Here are the results:

```
mean: 34124.91167 var: 0.07901676614
mean: 4.911666667 var: 0.07901666667
```

So the means are off by 34,120 but otherwise precisely identical (the .66666 would continue off the page if we let it), and the variances differ by one in the sixth nonzero digit, which is frankly not worth caring about. The ill-conditioning had no appreciable effect.

That, dear reader, is technological progress. Where a book from 2006 told us to take great care in implementing algorithms, all we had to do was throw twice as much space at the problem. If there's a speed difference between a program written with all `double`s and one written with all `float`s, I certainly can't perceive it, and it's worth extra seconds to be able to ignore so many caveats.

long long int Should we use `long int`s everywhere integers are used? The case isn't quite as open-shut. A `float` representation of π is more imprecise than a `double` representation of π, even though we're in the ballpark of three; both `int` and `long int` representations of numbers up to a few billion are precisely identical. The range of integers goes up to about $\pm$ 2.1 billion on a typical machine (I read that on some machines it can be scandalously short, like around 30,000 but I wonder if those are all obsolete at this point). If you think there's even a remote possibility that you have a variable that might multiply its way up to the billions (that's just

$200 \times 200 \times 100 \times 500$), then you certainly need to use a `long int` or even a `long long int`, or else your answer won't just be imprecise–it'll be entirely wrong, as C suddenly wraps around from +2.1 billion to -2.1 billion.

`long int`s aren't quite as immediate a drop-in replacement for `int`s as `double`s are for `float`s. I suppose `long int` looks ugly all over the place, though you can get away fine with just writing `long`, and you'll need to modify all your `printf`s to use `%li` instead of `%i`. Have a look at `/usr/include/limits.h` for details; on my machine it says that `int` and `long int` are actually identical.

But, again, if there actually is a cost to using `long`s and `long long`s with great frequency, it's darn cheap relative to the cost of going over the max and rolling over to a negative number.