

# R is slow

Eric Blair

30 March 2006

I find that people I talk to often don't realize how quickly their computer can do computations. Somebody doing analytic work will complain to me that they are using a nice, lean program to shunt data, but it still takes all night or even all week for their program to run; they presume that they need to buy better hardware, or tweak their code to run that extra 10% faster.

But it is the software that is slowing them down by orders of magnitude.

I had a chance to test the speed of R today. A user requested that I add a Fisher Exact test to Apophenia. Being a smarter programmer than mathematician, I knew to copy and paste the code from elsewhere.

So, I took the code from R. That is, Apophenia, a library of functions for doing stats in C, and R, a stats package, use exactly the same C code for doing this test. [There were a few tweaks to the code to de-Rify it, such as replacing `R_alloc` with plain old `malloc`, but about 98% of the code is identical.]

There is no better chance to test the two. The Fisher Exact test is a computationally-intensive activity, but both R and the C library are using the same algorithm. I wrote an R script and a C program to do a million Fisher Exact tests on a two by two matrix (the code is printed below) and used `time` to see how long they took. I made sure to not print anything to screen, and used the same matrix repeatedly because random number generation routines would vary.

Since the Fisher test uses factorials, there is much more number-crunching to be done when the figures are larger. So I tried one run with a two by two matrix with values in the tens, and one with values in the hundreds.

Here are the total times, and their ratios. s=seconds, m=minutes:

[Update, 12 January 2007:

This post recently got a lot of attention, and a few points of critique, the most salient being that `fisher.test` does some additional computationally-intensive work to produce additional statistics after calling the C routine. So in my efforts to run a fair fight, I produced an edited version of `fisher.test` that returns immediately after calling the C routine; if you repeat these timing tests with the stock version of the function, the R-to-C ratio will more than double.

I've thus deleted the the old timings, which put R in a worse light, and replaced them with more comparable numbers. I apologize if that means some pre-edit comments are now out of sync, but I felt this would be fairer than leaving up the old, worse-for-R numbers.

And to address one more hypothesis: no, this isn't about paging. I kept an eye on my hard drive light and it did nothing. I have done other timing tests where R did thrash the hard drive, on a problem where C code with no memory management managed to run just fine, but today's post is about processor speed, not memory use.]

C time	2m 49s
R time	88m 47s
R time/C time	31.5

As you can see, R didn't take longer by a few percent—it took about thirty times longer.

When I tell people that their work could be faster if they don't use a stats package, their first response is typically disbelief, thinking that the difference can only be a few percent. But even with a toy example like this, you're seeing a thirty-fold slowdown.

Their second response is to say that oh, they can just write C code to do the computationally intensive parts and call them from their favorite package. But here, I wrote an R script that does nothing but call C code over and over. This is where you'd end up, for example, if you had a simulation run from R with all the math being done in C. Many estimations of global parameters of a model would work like this. There are little data-shunting tasks on the R side surrounding the C function, and I could move those to the C side too, but at that point, why am I using R at all?

Now, R has its place: I'm not going to tell a statistics student or somebody doing exploratory poking around on a medium-sized data set to learn C. The code base is so complete, I often use it as a reference, as above. But nothing comes for free, and the cost of the convenience and general pleasantness of R is difficulty with large computations.

All this generalizes to nonscientific software as well: getting rid of the desktop manager (GNOME, KDE, or Microsoft Windows) makes a huge difference, and using a text editor instead of a GUI word processor is also night-and-day in terms of speed. But that's not the rant I'm writing today, because for user-at-desk applications you can basically do what you want with a slow program and at worst it's just a little more frustrating. But when a simulation, optimization search, or inversion of a large matrix takes several days instead of under an hour, there are techniques that one as a practical matter can not use. The science you do may actually suffer because of the tools you are using.

OK, now go read this textbook on statistical computing in C<sup>1</sup> and live a better life. Mr. GK of San Diego, CA did, and his analysis that took overnight to get 25% through in Matlab now runs in an hour. That person who requested a Fisher Exact test for Apophenia is somewhere not wasting hours right now. Art Garfunkel didn't switch to C, and his acting career never took off. I used to have a simulation written in R calling compiled C that took overnight to process 100 agents, but now that it's all in C simulations with 9,000 agents run in forty minutes. Don't risk it—learn to do statistical computing in C today!

**Details** For those of you who want to try this at home, here are the scripts I used. Readers who have other fave stats packages are encouraged to send in their own time comparisons with R or C, but note that some packages may approximate large Fisher tests via a chi-squared test. For the purposes here, they are cheating. [And once again, please note that I used a hacked version of the R-side `fisher.test` function.]

---

<sup>1</sup><http://modelingwithdata.org>

```
test_ct <- 5e6
x       <- matrix(c(30, 86, 24, 38), nrow=2)
for (i in 1:test_ct)
  {fisher.test(x)}
```

And the C code:

```
#include <apop.h>
int main(){
int      i, test_ct = 5e6
double   data[]     = { 30, 86, 24, 38 };
apop_data *testdata = apop_line_to_data(data,0,2,2);
  for (i = 0; i< test_ct; i++)
    apop_test_fisher_exact(testdata);
}
```

The reader well-versed with Apophenia will notice that there is a memory leak, because `apop_test_fisher_exact` returns an `apop_data` struct that never gets freed. But a million lost matrices didn't affect the speed of the program at all. The lesson from this is that the details of memory management that R is handling for you are not such a big deal on a modern PC anyway.