Drawing 10e12 samples from a judgmental estimation model

Nuño Sempere, Jorge Sierra

December 13, 2024

Abstract

To draw 10e12 samples you need two things: a fast language, and a supercomputer.

Background and motivation

Estimators in the tradition of Fermi or Tetlock sometimes create models that seek to capture their subjective credences—as opposed to, for instance, fitting a model to an underlying set of samples. One might go about this is by expressing a distribution using the set of idioms that grew from the foretold.io experimental forecasting platform into Squiggle—now a programming language for intuitive probabilitistic estimation. Some of the idioms in squiggle have been ported to Python (SquigglePy), to C (Squiggle.c), or to go.

However, those idioms favour relying on Monte Carlo estimation, that is, approximating a model by drawing samples rather than specifying the model exactly. This approach sometimes has difficulty modelling long-tail behaviour.

For instance, in our squiggle.c test code, we draw 10M samples from various distributions, and then compare the mean and standard deviation of the samples to the mean and standard deviation of the underlying distributions. This sometimes shows relative errors of 50% or more. For insance:

```
[-] Mean test for lognormal(0.644931, 4.795860) NOT passed.
Mean of lognormal(0.644931, 4.795860): 125053.904456,
  vs expected mean: 188163.894101
  delta: -63109.989645, relative delta: -0.504662
[-] Std test for lognormal(0.644931, 4.795860) NOT passed.
Std of lognormal(0.644931, 4.795860): 39976300.711166,
  vs expected std: 18577298706.170452
  delta: -18537322405.459286, relative delta: -463.707799
```

To explore those limits, and also just for the sake of the technical challenge, we describe how to draw 10e12 samples (an American trillion) from a judgmental model. In this case, the model is a very rough back-of-the-envelope estimate of the altruistic impact of Sentinel—a foresight and emergency response team set up by one of the authors—in basis points of existential/catastrophic risk[1] mitigated or averted per million dollars. We show how summary statistics change as we draw more samples. We conclude discussing limitations and practical applications.

1. How to draw 10e9 samples

1.1. Use a fast language.

Our time to botec repository compares how fast drawing 10e6 samples from a simple back-of-the-envelope calculation runs in different languages. The comparison depends on our familiarity with the different languages: as we are more familiar with a language, we grow capable of writing better & faster code in it. Nonetheless, speeds to get 10e6 samples in one author's computer are as follows:

Language	Time
C	6.20ms
squiggle.c	7.20ms
go	21.20ms
Nim	41.10ms
Lua (LuaJIT)	68.80ms
Python (numpy)	118ms
OCaml (flambda)	185ms
Squiggle (bun)	384ms
Javascript (node)	395ms
SquigglePy (v0.27)	1,542ms
R (3.6.1)	4,494ms
Python 3.9	11,909ms
Gavin Howard's bc	16,170ms

Readers might have expected this list to include languages such as Rust or zig. FORTRAN, Lisp, Haskell, Java, C#, C++, or Julia are also missing. So are faster options, like GPU or even assembly programming. This is because we are not very familiar with those languages, or because in our attempts to use them we found them too "clunky".

Still, the above table shows that the time it takes to run a simple Fermi model varies 3 to 4 orders of magnitude depending on the language choice (and skill at using it), and so language choice will be an important choice.

1.2. Use standard optimizations.

Once we have chosen a fast language like C, we can deploy some standard practices, like inlining functions, passing pointers not values, or packing structs

[1]with a conversion factor of existential risk being 100x worse than catastrophic risk.

into a single cache line. The difference between an initial approach and a final version with more care can be notable.

Using compiler flags can also give a noticeable speed bump, as can profiling. However, we didn't take advantage of this fully for drawing 10e12 samples because the compiler in our local machines (gcc/clang) was different than the compiler in the supercomputer we used for our final run (icc patched for usage with MPI).

A memorable moment was switching from antiquated libraries to reading the literature and simply baking in our own randomness primitives, mostly following the work of Marsaglia. Back in our javascript days, stlib.io had needless complexity because of the need to support incompatible types of javascript models. This is somewhat of a judgment call. Languages such as Rust, zig, or even go have some well-optimized mathematical primitives, and pulling them in might be the better choice.

Using a fast pseudorandomness generation function for a uniform distribution, and fast ways to go from a uniform to other distributions.

Also buys one conceptual clarity, few levels of abstraction.

1.3. Use single-core and multi-code parallelism

We further used OpenMP (for an 8x to 16x speedup if using that many thread on our local machines, but 64x if using the cores in the Finisterrae supercomputer), and MPI, for a speedup that ended up being roughly 4x (4 nodes, minus a bit of overhead) but which could be much scaled. For fast multi-core orchestration, our sense is that there aren't that many alternatives to MPI.

We could further get another 10x speedup by spending 10x as much time (60 hours instead of 6 hours); however, that quickly doesn't scale. One could also get another multiplier by using a more recent generation computer, but we didn't pursue that option, or it wasn't available to us.

When doing parallelism, one interesting trick is to structure arrays such that each thread is accessing different parts of an array simultaneously, but without cache conflicts and misses. One can do this by starting each thread far enough, but one can also pad the elements of an array such that cache conflicts can't happen, as follows:

```
#define CACHE_LINE_SIZE 64
typedef struct seed_cache_box_t {
    uint64_t seed;
    char padding[CACHE_LINE_SIZE - sizeof(uint64_t)];
    // Cache line size is 64 *bytes*, uint64_t is 64 *bits* (8 bytes). Different units!
} seed_cache_box;

// ...
```

```
seed_cache_box* cache_box = (seed_cache_box*)malloc(sizeof(seed_cache_box) * (size_t)n_threa
```

Throughout, we used C macros sparingly, but we did use them to allow MPI and non-MPI code to exist side by side, such that we could run the OpenMP side on our local machine, with an MPI "world" of only one machine.

1.4. Design for speed then compromise, rather than the reverse

Throughout this project, we didn't always make the speed-maximalist decision all the time. Some notable places where we didn't are as follows:

squiggle.c has a sample_mixture function which takes an array of function pointers:

```
double weights[] = { 0.3, 0.4, 0.3 };
double (*samplers[])(uint64_t*) = { sample_dist_1, sample_dist_2, sample_dist_3 };
double x = sample_mixture(samplers, weights, n_dists, seed);
```

This adds a level of indirection, and would be slower than directly writing:

```
double p = sample_uniform(0,1, seed);
double x;
if(p < p1){
  return sample_dist_1(seed);
} else if (p<p1+p2){
  return sample_dist_2(seed);
} else {
  return sample_dist_3(seed);
}
```

We further didn't use algebraic manipulations. For instance, multiplying two lognormal distributions, or two beta distributions together, produces a result in the same family. However, when expressing a judgmental model, it isn't parsimonious to group betas and lognormals together to allow for algebraic manipulations: doing so would make the judgmental model much less readable, and the underlying code messier and less general. We took a ~2-3x slowdown hit because of this.

Ultimately, there s a sweet spot in the (high user accessibility, high complexity, low speed) end of the spectrum: tools like causal or squiggle. There is also a sweet spot in the (low user accessibility, low complexity, high speed): the project we are exploring. With some of the lessons from writting squiggle.c, one of the authors created fermi, a command line read-evaluate-print loop (REPL) in go in which it is much faster to write small models, at the cost of being slower (while still being able to draw 100K samples instantaneuosly)

The punchline is that by paying attention to speed from the beginning and then dialing it down for some expressivity, usability gains, we end up with a much faster result, one that is able to better explore the long tail of outcomes due to being able to draw enough samples to meaningfully draw from it.

Go has almost none of these tricks.

Compiling on icc compiler. Requires not using useful gcc extensions, like nested functions.

Reader can find recipes in the makefile.

2. Results as samples increase

2.1. Mean and standard deviation

2.2. Minimum and maximum

3. An abuse of the underlying PRNG primitives?

Do the underlying pseudo-random number generator primitives we are using stand up to scaling to 1T samples? If you can draw 1T samples *at all*, you can draw 100K, 1M, 10M samples very fast.

Conclusion

Rethink Priorities' models puny struggled (https://forum.effectivealtruism.org/posts/pniDWyjc9vY5sjGre/reth
priorities-cross-cause-cost-effectiveness-model) to reach more than 150k samples. Eventually, with some optimizations, they moved to the billions. Much more is possible with a focus on speed.

Other titles: The virtue of fast Monte Carlo primitives: Drawing 1T samples and a REPL for rapid fermi iteration