**Introduction**

The [Kolmogorov distribution](https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Smirnov_test#Kolmogorov_distribution) (which I call F(x)) is as follows:

F(x) = \frac{\sqrt{2 \pi}}{x} \sum_{k = 1}^{\infty} e^{-(2k - 1)^2 \pi^2/(8x^2)}

There is no known simpler form and we have to work with this sum as it is. This is an infinite sum. How can we compute the value of this infinite sum numerically?

Naïvely we can do the following:

summand <- function(x, k) sqrt(2 \* pi)/x \* exp(-(2 \* k - 1)^2 \* pi^2/(8 \* x^2))

# Compute F(1)

sum(summand(1, 1:500))

[1] 0.7300003

In other words, sum up many of the terms and you should be close to the actual infinite sum.

This is a crude approach. The answer is not wrong (numerically) but certainly we should understand why adding up that many terms works. Also, we could have added more terms than necessary… or not enough.

So how can we compute this sum that guarantees some level of precision while at the same time not adding any more terms than necessary? Unfortunately I don’t recall how to do this from my numerical methods classes, but I believe I have found an approach that works well enough for my purposes.

**Geometric Sums**

An infinite sum is defined as the limit of a sequence of finite sums. Let S_n = \sum_{k = 0}^n a_k, where a_kis some sequence (we can have a_k = \frac{1}{k}or a_k = 2^{-k}, for example; in the case of the Kolmogorov distribution, we had a_k = \frac{\sqrt{2 \pi}}{x} e^{-(2k - 1)^2 \pi^2/(8x^2)}). Then, by definition, S_{\infty} = \lim_{n \to \infty} S_n.

My attempt to compute this sum simply amounts to trying to find an nsuch that the difference between S_nand S_{\infty}is no larger than machine precision: that is, I want \left| S_n - S_\infty \right| \leq \epsilonwhere \epsilonis the machine precision of the computer.

Since we don’t know what S_{\infty}is, we can instead decide that machine convergence occurs when \left| S_n - S_{n + 1}\right| \leq \epsilon; that is, when one summand and the next summand are numerically indistinguishable. Since S_{n + 1} - S_n = a_{n + 1}, this criterion is the same as requiring that \left| a_{n + 1}\right| \leq \epsilon.

Every sum that converges requires the condition a_k \to 0, so this criterion always yields an nthat gives “numerical convergence”. Of course, any Calculus II student who was paying attention in their class can tell you that not all infinite sums with summands going to zero converges, with the classic counterexample being the [Harmonic series](https://en.wikipedia.org/wiki/Harmonic_series_(mathematics)). So this approach would claim that sums that diverge are numerically convergent, which is bad. We cannot even expect this method to work in cases where the sum does converge, but it does so slowly (see a later example). However, in some cases this approach may be okay.

Take the case of a geometric series:

\sum_{k = 0}^{\infty} \rho^k

with \left| \rho \right|less than 1. These sums converge; in fact, mathematicians consider them as converging quickly. We also have a formula for what the sum is:

S_{\infty} = \frac{1}{1 - \rho}

After some algebra, we can quickly find a rule for determining how many summands we need to attain “numerical convergence”:

n \geq \frac{\log \epsilon}{\log \left| \rho \right|} + 2

We can see that in action with some R examples:

.Machine$double.eps # Numerical accuracy of this system

[1] 2.220446e-16

log(.Machine$double.eps)/log(0.5)

[1] 52

sum(0.5^(0:53))

[1] 2

# 2 is the correct answer, but the interpreter rounds its output; is the answer

# actually 2?

sum(0.5^(0:53)) - 2 == 0

[1] TRUE

sum(0.5^(0:52)) - 2 == 0

[1] FALSE

**Caveats**

This method, though, should be met with suspicion. For instance, it will not work for a slowly convergent sum. Take for example \sum_{k = 1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}. If you apply the above technique, then “numerical convergence” is achieved for n > \epsilon^{-1/2}. Not only is that a very large number, it won’t achieve our goal of good numerical accuracy.

.Machine$double.eps^(-1/2)

[1] 67108864

N <- 67108865

sum((1:N)^(-2)) # This may take a while

[1] 1.644934

sum((1:N)^(-2)) - pi^2/6

[1] -1.490116e-08

That difference is much larger than numerical accuracy.

In fact, the technique doesn’t always work for geometric sums either, as demonstrated by these examples.[1](https://ntguardian.wordpress.com/2018/09/10/naive-numerical-sums-in-r/#fn-3431-1)

sum(.99^(0:4000)) - 100

[1] -8.526513e-14

sum(.999^(0:40000)) - 1000

[1] -9.094947e-13

**Conclusion**

However, while this method cannot guarantee quick convergence or even convergence, I think it’s good enough for the sum I want to compute.

First, the sum converges more quickly than a geometric sum, as the summand decrease at a rate of O(\rho^{n^2})rather than O(\rho^n). Second, a method trying to attain numerical accuracy would need to be programmed, and if its implementation is written in R, that implementation will likely be much slower than simply using sum(), since the latter is implemented using fast C code. Such an implementation would have to be written from scratch in C++ using a tool such as **Rcpp**. One must wonder whether the tiny numerical efficiency and speed one might potentially gain are worth the work; if x is large it may be best to just round off the CDF at 1.

In the end, using the lessons learned above, I implemented the Kolmogorov distribution in the package I’m writing for my current research project with the code below.

pkolmogorov <- function(q, summands = ceiling(q \* sqrt(72) + 3/2)) {

sqrt(2 \* pi) \* sapply(q, function(x) { if (x > 0) {

sum(exp(-(2 \* (1:summands) - 1)^2 \* pi^2/(8 \* x^2)))/x

} else {

0

}})

}

pkolmogorov <- Vectorize(pkolmogorov, "q")

Numerical summation, as I mentioned above, is something I know little about, so I’d appreciate any readers with thoughts on this topic (and knowledge of how this is done in R) to share in the comments.